Tom Manual

April 07, 2006

This manual contains information for Tom version 2.3alpha.
This manual also exists in Postscript or pdf.

Tom is a language extension which adds new matching primitives to languages like C, Java, and Caml. Although rich and complex, Tom is not a stand-alone language: like a preprocessor, it strongly relies on the underlying language (C, Java, or Caml), called *host-language* in the following. To this language, Tom adds several constructs.

The main construct, %match, is similar to the match primitive found in functional languages: given an object (called subject) and a list of patterns-actions, the match primitive selects the first pattern that matches the subject and performs the associated action. The subject against which we match can be any object, but in practice, this object is usually a tree-based data-structure, also called *term* in the algebraic programming community.

The match construct may be seen as an extension of the classical switch/case construct. The main difference is that the discrimination occurs on a term and not on atomic values like characters or integers: the patterns are used to discriminate and retrieve informations from an algebraic data structure. Therefore, Tom is a good language for programming by pattern matching, and it is particularly well-suited for programming various transformations on trees/terms or XML data-structures.

Information on Tom is available at the Tom web page tom.loria.fr.

Contents

I Tutorial **5**

1 Basis **7**
1.1 Defining a data-structure . . . 7
1.2 Retrieving information in a data-structure . . . 8
1.3 Combining Tom and Java . . . 8
1.4 The "Hello World" example . . . 9
1.5 Transformation using Vas . . . 12

2 XML **15**
2.1 Manipulating Xml documents . . . 15
2.2 Building and sorting Xml/DOM documents . . . 20

3 Advanced features **23**
3.1 Transformation of user defined data-structures . . . 23

II Language **27**

4 Tom **29**
4.1 The core language . . . 29
4.2 Notations . . . 29
4.3 Tom constructs . . . 30
4.4 Tom signature constructs . . . 36

5 Gom **41**
5.1 Gom syntax . . . 41
5.2 Generated API . . . 42
5.3 Combining Gom with Tom . . . 44

6 Strategies **45**
6.1 Basic operators . . . 45
6.2 Basic operators with identity considered as failure . . . 45
6.3 Some basic strategies . . . 46
6.4 Some basic strategies with identity considered as failure . . . 46
6.5 Using MuTraveler in Tom . . . 46
6.6 Strategy construct . . . 47

7 Runtime Library **49**

8 Migration Guide **51**
8.1 Migration from 2.2 to 2.3 . . . 51
8.2 Migration from 2.1 to 2.2 . . . 51
8.3 Migration from 2.0 to 2.1 . . . 54
8.4 Migration from 1.5 to 2.0 . . . 54

III Tools **57**

9 Installation **59**
9.1 Requirements . . . 59
9.2 Windows . . . 59
9.3 Unix . . . 59
9.4 Eclipse plugin . . . 60
9.5 Getting some examples . . . 60

10 Using Tom **61**
10.1 The Tom compiler . . . 61
10.2 Development process . . . 61
10.3 Command line tool . . . 61
10.4 Ant task . . . 63

11 Using Gom **65**
11.1 Command line tool . . . 65
11.2 Ant task . . . 65

12 Configuring your environment **67**
12.1 Editor . . . 67
12.2 Shell . . . 67

Part I

Tutorial

Chapter 1

Basis

1.1 Defining a data-structure

One of the most simple Tom program is the following:

```
import main.peano.types.*;

public class Main {
  %gom {
    module Peano
    abstract syntax
    Nat = zero()
        | suc(pred:Nat)
        | plus(x1:Nat, x2:Nat)
   }

  public void run() {
    Nat z    = `zero();
    Nat one  = `suc(z);
    System.out.println(z);
    System.out.println(one);
  }

  public final static void main(String[] args) {
    Main o = new Main();
    o.run();
  }
}
```

The %gom{ ... } constructs defines a data-structure, also called algebraic data-type. This data-structure is composed of a sort (Nat), and three operators (zero, suc, and plus). These operators are called constructors, because they are used to construct the data-structure. zero is a constant (arity 0), whereas the arity of suc and plus is respectively 1 and 2. In Gom, a name has to be given for each slot (pred for suc, and x1, x2 for plus).

A second construct provided by Tom is the ` (backquote), which builds an object. The expression `zero() builds the object zero() of sort Nat. Therefore, the variable z is a Java object that can be used to build the object suc(zero()). In other words, the definition of an algebraic data-structure can be seen as a grammar. The ` construct allows to build trees (Abstract Syntax Trees) over this grammar.

The implementation generated by Gom is such that objects can be converted into readable strings, using the toString() method.

We assume that Tom is installed, as explained in chapter 9. The command tom has to be used to compile the file Main.t. It will generate the file Main.java, that can be compiled and executed using javac and java:

```
> tom Main.t
> java Main.java
> java Main
zero
suc(zero)
```

1.2 Retrieving information in a data-structure

In addition to %gom and ‘, Tom provides a third construct: %match. Using the same program, we add a method evaluate(Nat n) and we modify the method run():

```
public Nat evaluate(Nat n) {
  %match(Nat n) {
    plus(x, zero()) -> { return ‘x; }
    plus(x, suc(y)) -> { return ‘suc(plus(x,y));}
  }
  return n;
}

public void run() {
  Nat two = ‘plus(suc(zero()),suc(zero()));
  System.out.println(two);
  two = evaluate(two);
  System.out.println(two);
}
```

The %match construct is similar to switch/case in the sense that given a tree (generally called subject), the %match selects the first pattern that *matches* the tree, and executes the associated action. By *matching*, we mean: finding the instances of variables that make the pattern and the subject identical. In our example, the first pattern matches the subject n when n is a tree rooted by plus, and whose second subterm corresponds to the tree zero(). When it is the case, the variable x takes the first subterm as value. We say that a tree is *rooted* by a symbol when its top-node corresponds to this symbol. when a pattern matches, we say that its variables are *instantiated*. They can then be used in the action part (also called right-hand side). Similarly, the second rules can be applied when the second subterm is rooted by a suc. In that case, x and y are instantiated.

The method evaluate defines the addition of two integers represented by Peano successors (i.e. zero and suc). The first case says that the addition of an integer with zero is evaluated into the integer itself. The second case builds a new term (rooted by suc) whose subterm (plus(x,y)) denotes the addition of x and y.

When executing the run() method, we obtain:

```
plus(suc(zero),suc(zero))
suc(plus(suc(zero),zero))
```

Note that in a match construct, the variables should not be declared by the programmer. They are written without (), and their type is inferred. A variable introduced should be *fresh*: it should not be declared earlier in a wrapping block, either in Java, either in a %match construct. A Tom variable can be used in the corresponding action, but has to be wrapped in a ‘ construct. Note that the scope of a back-quote expression is delimited by well-parenthesis sentences (‘zero(), ‘suc(zero()), ‘suc(x)). The special case ‘x, corresponding to a variable alone, is also correct. To write complex expressions, extra parenthesis can be added: ‘(x), ‘(x == y) for example.

1.3 Combining Tom and Java

As illustrated below, Tom offers a simple way to retrieve information stored in subterms. The previous example describes a single-step when adding two integers. To get a canonical form (a term that can no

longer be reduced), we need to encode a loop, to reach a fix-point. We also have to allow reductions in sub-trees (i.e. a plus which is not at a top-position).

```
public Nat evaluate(Nat n) {
%match(Nat n) {
      plus(x, zero()) -> { return `x; }
      plus(x, suc(y)) -> { return `suc(plus(x,y));}
// congruence rules
      suc(x)          -> { return `suc(evaluate(x));}
      plus(x,y)       -> { return `plus(evaluate(x),evaluate(y));}
}
return n;
}

  public void run() {
Nat t = `plus(suc(zero()),suc(zero()));
    System.out.println(t + " -> " + evaluate(t));
t = `plus(suc(zero()),plus(suc(zero()),suc(zero())));
    System.out.println(t + " -> " + evaluate(t));
  }
```

By adding two new congruence rules, a term that cannot be reduced by the first two rule will be recursively inspected by the two last rules. Therefore, if a transformation can be performed (at top position, or in sub-trees), it will be done.

```
plus(suc(zero),suc(zero)) -> suc(plus(suc(zero),zero))
plus(suc(zero),plus(suc(zero),suc(zero))) -> plus(suc(zero),suc(plus(suc(zero),zero)))
```

We use Java (while-loop or recursion) to compute a fix-point:

```
public Nat loop(Nat n) {
Nat tmp = `evaluate(n);
if(tmp==n) {
return n;
}
return loop(tmp);
}

  public void run() {
Nat t = `plus(suc(zero()),suc(zero()));
    System.out.println(t + " -> " + loop(t));
t = `plus(suc(zero()),plus(suc(zero()),suc(zero())));
    System.out.println(t + " -> " + loop(t));
  }
}
```

And we get the expected results:

```
plus(suc(zero),suc(zero)) -> suc(suc(zero))
plus(suc(zero),plus(suc(zero),suc(zero))) -> suc(suc(suc(zero)))
```

1.4 The "Hello World" example

One of the most simple Tom program is the following:

```
public class HelloWorld {
  %include { string.tom }
```

```
  public String getWord(String t) {
    %match(String t) {
      "World"  -> { return "World";}
      _        -> { return "Unknown"; }
    }
  }

  public final static void main(String[] args) {
    HelloWorld o = new HelloWorld();
    System.out.println("Hello " + o.getWord("World"));
  }
}
```

The %include { string.tom } construct imports the predefined (Java) mapping which defines the Tom String algebraic sort. Thus, String can be used in the %match(String t) construct.

This examples shows that pattern matching can be performed against built-in data-types. In this example, the %match construct is equivalent to a switch/case construct. '_' denotes an anonymous variable that corresponds to a default case in this example.

1.4.1 "Hello World" revisited

One particularity of Tom is to support list-matching (also known as associative-matching).

When considering that a string is a list of characters, it becomes natural to describe pattern-matching using an associative operator which corresponds to the concatenation of two characters. Thus, the string "Hello" is equivalent to the list of characters ('H','e','l','l','o').

Given a string, to check if the string contains the character 'e', we can use the following matching construct:

```
%match(String t) {
  (_*,'e',_*) -> { System.out.println("we have found a 'e'"); }
}
```

In this example, '_*' corresponds to an anonymous variable which can be instantiated by any list of characters (possibly reduced to the empty list).

In order to capture the context, it is possible to use a named variable:

```
%match(String t) {
  (before*,'e',after*) -> {
      System.out.println("we have found a 'e'" +
                         " after " + `before* +
                         " but before " + `after*);
  }
}
```

In this example, we have introduced two new variables (before* and after*), called "variable-star", which are instantiated by a list of characters. This is '*' that denoted a list variable.

To use their value in a Java expression, Tom provides the '`' (backquote) mechanism. This construct can be use to retrieve the value of a Tom variable (instantiated by pattern-matching). As illustrated by the previous examples, `before* and `after* are replaced by expressions of sort String.

Suppose now that we look for a word whose last letter is a 'o'. A possible Tom pattern is:

```
%match(String t) {
  (before*,'o') -> { ... }
}
```

Using this mechanism, it also becomes possible to look for a word which contains an 'e' somewhere and an 'o' in last position:

```
%match(String t) {
  (before*,'e',_*,'o') -> { ... }
}
```

Note that some pattern could provide several outcomes:

```
%match(String t) {
  (before*,'l',after*) -> {
      System.out.println("we have found " + ‘before* +
                         " before 'l' and " + ‘after* + "after");
  }
}
```

It will match twice:

```
we have found he before 'l' and lo after
we have found hel before 'l' and o after
```

Let us suppose that we look for two consecutive ‘l’ anywhere in the matched expression. This could be expressed by:

```
%match(String t) {
  (_*,'l','l',_*) -> { ... }
}
```

Since this syntax is error prone when looking for long/complex substrings, Tom provides an abbreviated notation: ‘ll’:

```
%match(String t) {
  (_*,'ll',_*) -> { ... }
}
```

This notation is fully equivalent to the previous one.

1.4.2 “Hello World” re-revisited

As illustrated previously, we can use variables to capture contexts. In fact, this mechanism is more general and we can use a variable anywhere to match something which is not statically know. Thus, it is possible to use variable (without a star) to explicitly say that a character should be there:

```
%match(String t) {
  (x,_*,'ll',_*,y) -> { ... }
}
```

This example matches the string t if this string contains two consecutive ‘l’ and at least two other characters. In this case, the first letter of the string is stored in the variable ‘x’ (respectively the last letter in ‘y’).

The variable ‘x’ and ‘y’ are local to the given pattern. Thus, when considering several patterns in a same match construct, there is no interference between patterns. However, in a same pattern, we can use two times a same variable to indicate to we want to match two identical elements. This feature is known as non-linear matching:

```
%match(String t) {
  (x,_*,'ll',_*,y) -> { ... }
  (x,y,y,x)        -> { /* we have found a palindrome */ }
}
```

When looking for several instances of a same character, it may be interesting to use a variable to describe this repetition:

```
%match(String t) {
  (_*,'a',_*,'a',_*,'a',_*) -> { /* look for 3 'a' in a string */ }
  (_*,x@'a',_*,x,_*,x,_*)   -> { /* look for 3 'a' in a string */ }
}
```

Here, the first 'a' is bound to 'x', and then, all remaining occurrences of 'x' are compared to this first instance. 'x' is an alias for ''a''.

In the previous section, we have said that (_*,'ll',_*) is equivalent to (_*,'l','l',_*). This is still true, but the reader should also note that (_*,x@'ab',_*) becomes (after expansion) equivalent to (_*,x@'a',x@'b',_*), which can never match any string.

1.5 Transformation using Vas

On one side Tom offers customized pattern matching facilities to support the development of XML and string based applications. On the other side, Tom is a rule based engine which allows to solve complex pattern matching problems over algebraic data-types.

In order to describe user defined abstract data-types, Tom provides a signature definition mechanism (%typeterm, %typelist, %op, *etc.* c.f. section 3.1). However, Tom does not provides any support to implement these abstract data-types. This is why, in addition to Tom, we have developed the system Vas. This tool provides a human readable syntax definition formalism (inspired from SDF) as well as an efficient implementation for the data-type. The implementation is generated by Apigen and based on the ATerm library, using maximal sharing in an efficient way.

In the following, we consider a simple abstract data-type which defines three sorts (Date, Person, and PersonList), as well as three constructors (date, person, and concPerson). In this signature, Int and String are predefined builtin sorts:

```
%vas {
  module person
    imports int str

  public
    sorts Date Person PersonList

  abstract syntax
    date(day:Int, month:Int, year:Int)                          -> Date
    person(firstname:String, lastname:String, birthdate:Date) -> Person
    concPerson(Person*)                                         -> PersonList
}
```

The notation concPerson(Person*) means that concPerson is a variadic associative operator which takes a list of Person in arguments, and returns a PersonList.

By using the %vas{ ... } construct, a Tom signature and an implementation are automatically generated. In order to connect these generated files, it is only necessary to import the correspondings packages, define the getPersonFactory() function and initializing the instance of the factory The name of this function depends on the name of the Vas module (person in this example).

```
package tutorial;
import aterm.*;
import aterm.pure.*;
import tutorial.addressbookvas.person.*;
import tutorial.addressbookvas.person.types.*;

public class AddressBookVas {
  %vas { ... }

  private personFactory factory;
  public AddressBookVas(personFactory factory) {
```

```
    this.factory = factory;
  }

  public personFactory getPersonFactory() {
    return factory;
  }

  public final static void main(String[] args) {
    AddressBookVas test = new AddressBookVas(personFactory.getInstance(new PureFactory()));
    test.run();
  }
```

Once importation, definition and initialization are performed, it becomes very easy to use Tom: the constructors can be immediately used in a backquote or a matching constructs.

1.5.1 Using the backquote mechanism

In this section, we consider a tiny address-book application: given a date and a list of persons, the problem consists in finding out the persons whose birthday corresponds to the date.

The first sub-problem consist in building a list of persons. For this, we use the backquote notation and the concPerson operator:

```
  public PersonList generateBook() {
    return `concPerson(
      person("John","Smith",date(1965,3,27)),
      person("Marie","Muller",date(1986,3,26)),
      person("Paul","Muller",date(2000,1,27))
      );
  }
```

1.5.2 Using list-matching facilities

The run() method generates a list of persons, builds a date object and calls the happyBirthday() method whose goal is to determine persons whose birthday corresponds to the date:

```
  public void run() {
    PersonList book = generateBook();
    Date today = `date(2003,3,27);
    happyBirthday(book,today);
  }
```

The following method is interesting because it uses pattern matching to encode a searching procedure. As illustrated below, the use of associative operator, combined with non-linearity, allows us to solve the problem in a nice and concise way:

```
  public void happyBirthday(PersonList book, Date date) {
    %match(PersonList book, Date date) {
      concPerson(_*, person(firstname, _, date(year,month,day)), _*), date(_,month,day)   -> {
        System.out.println("Happy birthday " + `firstname);
      }
    }
  }
```

Let us mention that the println statement is executed for all persons whose birth-date corresponds to the given date. It also interesting to note that all instantiations of firstname, year, month and day are tried before exiting the %match construct.

Chapter 2

XML

Even if this section mostly deals with XML features, every explained technics can be used with other data-structures.

2.1 Manipulating Xml documents

This example is inspired from a practical study. In the following, we consider a simple modelisation of the Platform for Privacy Preferences Project (P3P), developed by the World Wide Web Consortium, which is emerging as an industry standard providing a simple, automated way for users to gain more control over the use of personal information on Web sites they visit.

Given a client and a server, the problem consists in verifying that the client's policy is compliant with the server's policy. Both policies preferences are expressed in APPEL (A P3P Preference Exchange Language) and written in Xml. The server's policy is the following:

```
<POLICIES xmlns="http://www.w3.org/2002/01/P3Pv1">
  <POLICY name="mypolicy" discuri="http://www.ibm.com/privacy"
          opturi="http://www.ibm.com/privacy" xml:lang="en">
    <STATEMENT>
      <RECIPIENT> <delivery/> </RECIPIENT>
      <PURPOSE> <contact/> </PURPOSE>
      <DATA-GROUP>
        <DATA ref="#dynamic.clickstream"/>
        <DATA ref="#dynamic.http"/>
        <DATA ref="#dynamic.clientevents"/>
        <DATA ref="#user.home-info.postal.country"/>
        <DATA ref="#dynamic.cookies"> <CATEGORIES> <content/> </CATEGORIES> </DATA>
      </DATA-GROUP>
    </STATEMENT>
  </POLICY>
</POLICIES>
```

The client's policy is the following:

```
<RULESET appel="http://www.w3.org/2002/04/APPELv1"
         p3p="http://www.w3.org/2000/12/P3Pv1"
         crtdby="W3C" crtdon="2001-02-19T16:21:21+01:00">
  ...
  <RULE behavior="limited1" prompt="yes">
    <POLICY>
      <STATEMENT>
        <PURPOSE connective="and-exact"> <current/> </PURPOSE>
        <RECIPIENT connective="or">
          <other-recipient/>
```

```
          <public/>
          <unrelated/>
        </RECIPIENT>
      </STATEMENT>
    </POLICY>
  </RULE>
  ...
  <RULE behavior="request" prompt="yes">
    <POLICY>
      <STATEMENT>
        <DATA-GROUP>
          <DATA ref="#dynamic.clientevents"> </DATA>
          <DATA ref="#dynamic.clickstream"/>
        </DATA-GROUP>
      </STATEMENT>
    </POLICY>
  </RULE>
  ...
</RULESET>
```

For expository reasons, we consider only a sub-problem in which we say that a client's policy is compatible with the server's policy if all ref attributes from a <DATA></DATA> node also appear on the server side. In the considered examples, the policies are compatible because

```
<DATA-GROUP>
  <DATA ref="#dynamic.clientevents"> </DATA>
  <DATA ref="#dynamic.clickstream"/>
</DATA-GROUP>
```

is included in:

```
<DATA-GROUP>
  <DATA ref="#dynamic.clickstream"/>
  <DATA ref="#dynamic.http"/>
  <DATA ref="#dynamic.clientevents"/>
  <DATA ref="#user.home-info.postal.country"/>
  <DATA ref="#dynamic.cookies"> <CATEGORIES> <content/> </CATEGORIES> </DATA>
</DATA-GROUP>
```

The problem consists in implementing such a verification tool in Tom and Java.

2.1.1 Loading Xml documents

The first part of the program declares imports, and defines the main and the run methods.

```
import tom.library.xml.*;
import tom.library.adt.tnode.*;
import tom.library.adt.tnode.types.*;
import aterm.*;
import java.util.*;

public class Evaluator {

  %include{ adt/tnode/TNode.tom }

  private XmlTools xtools;
  private TNodeFactory getTNodeFactory() {
    return xtools.getTNodeFactory();
```

```
}

public static void main (String args[]) {
  Evaluator evaluator = new Evaluator();
  evaluator.run("server.xml","client.xml");
}

private void run(String serverfile,String clientfile){
  xtools = new XmlTools();
  TNode server = (TNode)xtools.convertXMLToATerm(serverfile);
  TNode client = (TNode)xtools.convertXMLToATerm(clientfile);
  boolean compatible = compareDataGroup(getDataGroup(server.getDocElem()),
                                        getDataGroup(client.getDocElem()));
  System.out.println("result = " + compatible);
}
```

As explained in the TOM Reference Manual, to each XML document corresponds a DOM (Document Object Model) representation. In Tom, we have defined a mapping from DOM sorts to abstract algebraic sorts: TNode and TNodeList, which correspond respectively to Node and NodeList, defined by the Java DOM implementation.

This mapping has to be initialized in two different ways:

- the importation of tom.adt.tnode.* and tom.adt.tnode.types.* defines the sorts TNode and TNodeList and allows us to build objects over these two sorts.
- the %include{ TNode.tom } Tom construct is similar to the #include C preprocessor construct. In this case, it imports the definition of Tom algebraic constructors needed to manipulate XML documents.

Since Apigen (the tool used to generate concrete implementations of abstract data-types) is completely independent of Tom, an access function (getTNodeFactory) has to be defined. The name of this function is related to the name of the abstract data-type (TNode in our case).

In complement to Tom, we provide a runtime library which contains several methods to manipulate XML documents. In particular, we provide the XmlTools class (defined in the jtom.runtime.xml.* package).

The run method takes two filenames as arguments, reads the associated files and convert their XML content into TNode terms (using the convertXMLToATerm function).

The getDataGroup function is used to retrieve a <DATA></DATA> node in the considered XML document. Note that getDocElem is first applied to XML documents in order to consider subterms of the DocumentNode element. Then, the compareDataGroup function is used to check that the client's policy is compatible with the server's policy.

2.1.2 Retrieving information

Given an XML subtree, the problem consists in extracting a <DATA-GROUP></DATA-GROUP> node. For expository reasons, we first consider that such a <DATA-GROUP></DATA-GROUP> node can only appear at two different specific places: one for the client's definition and one for the server's definition.

Thus, in the Tom program we naturally consider two cases:

```
private TNode getDataGroup(TNode doc) {
  %match(TNode doc) {
    <POLICIES>
       <POLICY>
         <STATEMENT>
           datagroup@<DATA-GROUP></DATA-GROUP>
         </STATEMENT>
       </POLICY>
```

```
    </POLICIES> -> { return datagroup; }

    <RULESET>
      <RULE>
        <POLICY>
          <STATEMENT>
            datagroup@<DATA-GROUP></DATA-GROUP>
           </STATEMENT>
        </POLICY>
      </RULE>
    </RULESET> -> { return datagroup; }
  }

  return `xml(<DATA-GROUP/>);
}
```

The first pattern means that a <DATA-GROUP></DATA-GROUP> node has to be found under a <STATEMENT></STATEMENT> node, which should be under a <POLICY></POLICY> node, which should be under a <POLICIES></POLICIES> node. Once such a pattern is found, the mechanism allows us to give a name (datagroup) to this subterm and reuse it in the action part: return datagroup;.

As explained in the TOM Reference Manual, the XML notation implicitly extends the given patterns by adding context variables. Thus, the <DATA-GROUP></DATA-GROUP> pattern means that a subterm headed by <DATA-GROUP></DATA-GROUP> is searched. But this subterm can contain attributes and children even if it is not explicitly defined by the notation. To make the definition of patterns more precise, the user can use the explicit notation and define the following pattern: <DATA-GROUP (_*)>(_*)</DATA-GROUP>.

Note that a Node object is a ternary operator Element whose first subterm is the name of the XML node, the second subterm is a list of attributes and the third subterm is a list of subterms (which correspond to XML sub-elements). The second and the third elements are terms of sort TNodeList (because they are implemented by NodeList objects in DOM).

2.1.3 Comparing two Xml subtrees

Given two <DATA-GROUP></DATA-GROUP> nodes, the problem consists in checking that all <DATA></DATA> nodes from the first tree also appear in the second one. This can be easily implemented by the following couple of functions:

```
private boolean compareDataGroup(TNode server, TNode client) {
  boolean res = true;
  %match(TNode client) {
    <DATA-GROUP><DATA ref=reftext></DATA></DATA-GROUP>
    -> { res = res && appearsIn(reftext,server); }
  }
  return res;
}
```

Given a <DATA-GROUP></DATA-GROUP> tree, we look for a <DATA></DATA> subtree which contains an attribute named ref. When such an attribute exists, its content (a string) is stored into the Java reftext variable. The appearsIn function is then used to check that the attribute also appears on the server side.

```
private boolean appearsIn(String refclient, TNode server) {
  %match(TNode server) {
    <DATA-GROUP><DATA ref=reftext></DATA></DATA-GROUP>
    -> {
       if(reftext.equals(refclient)) {
```

```
          return true;
        }
      }
    }
    return false;
  }
```

This piece of code is interesting because it introduces conditional rules. Given the string refclient we look for a <DATA></DATA> subtree containing a ref attribute with exactly the same string. Since it is not possible to use an instantiated variable in a pattern (something like <DATA ref=refclient>...</DATA>, we have to introduce a fresh variable reftext and check in a condition that this variable is equal to refclient. This is done via a Java condition: the action part (return true;) is executed only when the condition is satisfied. Note that, as in a switch/case statement, when the action part is not exited by a return, break or goto, the control flow is transferred to the next matching solution or the next matching pattern. In this case, another substitution has to be searched (i.e. another <DATA></DATA> subtree which contains a ref attribute).

If no such subtree exists, this means that the two policies are not compatible and false is returned.

2.1.4 Retrieving information using traversal functions

The Tom runtime library provides a set of generic functions in order to perform various kinds of traversal over a term. This is useful when searching a pattern somewhere in a term, at any position.

In the current example, the getDataGroup functions looks for a <DATA-GROUP></DATA-GROUP> node. Instead of statically specifying the path (POLICIES, POLICY, STATEMENT, DATA-GROUP and RULESET, RULE, POLICY, STATEMENT, DATA-GROUP) we could have used the generic traversal mechanism provided by the package jtom.runtime.*.

```
private GenericTraversal traversal;
public Evaluator() {
  this.traversal = new GenericTraversal();
}
```

In the following, we consider a variant of the previous problem where we want to be able to consider all the <DATA-GROUP></DATA-GROUP> nodes which appear in an XML document.

The following method uses the generic traversal mechanism. Given a Collection and a term to inspect (called subject), the collectDatagroup method creates an inner class (Collect1 where 1 stands for the number of arguments of the apply method) and applies the collect object to all subtrees of the subject term.

```
protected void collectDatagroup(final Collection collection, TNode subject) {
  Collect1 collect = new Collect1() {
      public boolean apply(ATerm t) {
        if(t instanceof TNode) {
          %match(TNode t) {
            <DATA-GROUP> </DATA-GROUP> -> {
               collection.add(t);
               return false;
             }
          }
        }
        return true;
      } // end apply
    }; // end new

  traversal.genericCollect(subject, collect);
}
```

When considering the apply method: we have to imagine that this method is applied to all subterms of the subject term. Since a term may contain subterms of different sorts (integers or strings for example), the instanceof construct is used as a first filter to select only the subterms which correspond to XML nodes. When such a subterm is rooted by a <DATA-GROUP></DATA-GROUP>, the subterm is added to the collection (by a side-effect). The return false; statement indicates that it is no longer necessary to traverse the considered term (in our example, a <DATA-GROUP></DATA-GROUP> node cannot be found inside a <DATA-GROUP></DATA-GROUP> itself). Respectively, the return true; statement indicates that the apply function should be recursively applied to the current term in order to continue the traversal.

Given a collection, the getDataGroup method uses an iterator to select a <DATA-GROUP></DATA-GROUP> node.

```
private TNode getDataGroup(TNode doc) {
  HashSet c = new HashSet();
  collectDatagroup(c,doc);
  Iterator it = c.iterator();
  while(it.hasNext()) {
    TNode datagroup = (TNode)it.next();
    return datagroup;
  }

  return `xml(<DATA-GROUP/>);
}
```

2.2 Building and sorting Xml/DOM documents

In this section, we consider a DOM mapping for XML documents. This means that XML documents are still considered as algebraic terms (built over TNode and TNodeList), but their internal representation is backed by the W3C DOM library.

In the following, we consider a small data-base represented by an XML document:

```
<Persons>
  <Person Age="30"> <FirstName> Paul   </FirstName> </Person>
  <Person Age="28"> <FirstName> Mark   </FirstName> </Person>
  <Person Age="21"> <FirstName> Jurgen </FirstName> </Person>
  <Person Age="21"> <FirstName> Julien </FirstName> </Person>
  <Person Age="24"> <FirstName> Pierre-Etienne </FirstName> </Person>
</Persons>
```

The problem consists in sorting this document according to different criteria (age or first-name for example).

2.2.1 Loading Xml documents

The first part of the program declares imports, and defines the main and the run methods.

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.File;

public class PersonSort1 {
  private Document dom;

  %include{ dom.tom }
```

```
public static void main (String args[]) {
  PersonSort1 person = new PersonSort1();
  person.run("person.xml");
}

private void run(String filename){
  try {
    dom = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(filename);
    Element e = dom.getDocumentElement();
    dom.replaceChild(sort(e),e);

    Transformer transform = TransformerFactory.newInstance().newTransformer();
    StreamResult result = new StreamResult(new File("Sorted.xml"));
    transform.transform(new DOMSource(dom), result);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

The mapping has to be initialized in the following ways:

- the importation of org.w3c.dom.* and javax.xml.* packages in order to use the DOM library.
- the %include{ dom.tom } construct imports the definition of Tom algebraic constructors needed to manipulate DOM objects.
- the dom variable has to be instantiated. This variable corresponds to the notion of Document in the DOM terminology.

2.2.2 Comparing two nodes

In order to implement a sorting algorithm, we need to know how to compare two elements.

The following function takes two XML documents in argument and compares their attribute Age, using the string comparison operator compareTo.

```
private int compare(Node t1, Node t2) {
  %match(TNode t1, TNode t2) {
    <Person Age=a1></Person>, <Person Age=a2></Person> -> { return ‘a1.compareTo(‘a2); }
  }
  return 0;
}
```

In this example, it is interesting to note that an XML node is seen as an associative operator. This feature, combined with the implicit notation, is such that <Person Age=a1></Person> will match any XML node headed by Person which contains the attribute Age. This XML node may contain several sub-nodes, but they will not be stored in any variable.

2.2.3 Sorting a list of nodes

In this section we use a swap sort algorithm which consists in finding two elements that are not in the correct order. Once they are found, they are swapped before calling the algorithm recursively. When no two such elements exist, this means that the list is sorted.

```
private Node sort(Node subject) {
  %match(TNode subject) {
   <Persons>(X1*,p1,X2*,p2,X3*)</Persons> -> {
      if(compare(‘p1,‘p2) > 0) {
```

```
          return sort(`xml(dom,<Persons>X1* p2 X2* p1 X3*</Persons>));
        }
      }

      _ -> { return subject; }
    }
  }
```

The pattern <Persons>(X1*,p1,X2*,p2,X3*)</Persons> looks for two elements p1 and p2 and stores the remaining contexts in X1*, X2*, and X3*. In order to give a name to contexts (and then retrieve them to build a new list), the pattern is written in explicit notation. This notation prevents Tom to add extra variables. Otherwise, the expanded form of <Persons>X1* p1 X2* p2 X3*</Persons> (written in explicit notation) would have been <Persons>(_*,X1*,_*,p1,_*,X2*,_*,p2,_*,X3*,_*)</Persons>.

Note that the action part is guarded by a condition (if(compare(p1,p2) > 0)). This means that return sort(...) is executed only when two bad-ordered elements are found. When p1 and p2 are correctly ordered, the matching procedure continues and extracts another couple p1 and p2. As mentioned in the TOM Reference Manual, when Tom cannot find two elements p1 and p2 such that the condition is satisfied (the list is sorted), the return sort(...) statement is not executed and the control flow is transferred to the following pattern: _ -> { return subject; }

As mentioned previously, when two elements have to be swapped, a new list is built. This done via the ‘‘’ (backquote) construct. This construct, used before to retrieve instantiated variables, can also be use to build a new term. In order to build an XML document, the xml(...) function has to be used. The number of parameters depends on the XML mapping which is used. In our case, when manipulating DOM objects, a new subtree is built is the context of a main document. This extra-parameter is given in the first argument of the xml function. Thus, ‘xml(dom,<Persons>...<Persons>) builds a new <Person></Person> node, as a child of the dom document. When using the TNode.tom mapping, a library based on Apigen is used. In this case, it is no longer necessary (and even not correct) to use this extra argument.

2.2.4 Sorting by side effect

In the previous section, we have considered a sorting algorithm expressed in a pure functional programming style: when two elements have to be swapped, a new list is built and returned.

In the following example, we exploit OO-style of the DOM library and perform a swap in place: the list is updated by swapping two elements (with side effect):

```
private void sort(Node subject) {
  %match(TNode subject) {
    r @ <_>p1@<_ Age=a1></_> p2@<_ Age=a2></_></_> -> {
      if(`a1.compareTo(`a2) > 0) {
        `r.replaceChild(`p2.cloneNode(true),`p1);
        `r.replaceChild(`p1,`p2);
        `sort(r);
        return;
      }
    }
  }
}
```

In this example, we use the notion of anonymous XML node (<_>...</_>) which allows to describe more generic algorithms.

Chapter 3

Advanced features

3.1 Transformation of user defined data-structures

Up to now, we have considered examples where the implementation of data-types is either predefined (XML case), or generated by a tool (Vas case). One interesting contribution of Tom is to be customizable. By defining a mapping between the signature formalism (%typeterm, %op, *etc.*) and the concrete implementation, it becomes possible to perform pattern matching against any data-structure.

3.1.1 Defining a simple mapping

In this section, we consider that we have a Java library for representing tree based data structures. On another side, to be able to use Tom, we consider the following abstract data-type:

```
%typeterm Nat
%op Nat zero
%op Nat suc(Nat)
```

By doing so, we have defined a sort (Nat), and two operators (zero and suc). When defining a mapping, the goal consists in explaining to Tom how the considered abstract data-type is implemented, and how a term (over this signature) can be de-constructed.

For expository reasons, the ATerm library is the Java library used to implement terms. However, any other tree/term based library could have been used instead.

In order to define a correct mapping, we have to describe how the algebraic sort (Nat in our example) is implemented (by the ATermAppl class). This is done via the implement construct:

```
%typeterm Nat {
  implement { ATermAppl }
}
```

The second part of the mapping definition consists in defining how the symbols (zero and suc) are represented in memory. This is done via the is_fsym construct:

```
%op Nat zero {
  is_fsym(t) { t.getName().equals("zero") }
}

%op Nat suc(pred:Nat) {
  is_fsym(t)       { t.getName().equals("suc") }
  get_slot(pred,t) { (ATermAppl)t.getArgument(0) }
}
```

In addition, get_slot describes how to retrieve a subterm of a term.

Given a term built over the ATerm library, it becomes possible to perform pattern matching as previously explained.

3.1.2 Using backquote constructs

When using the backquote construct (‘suc(suc(zero)) for example), Tom has to know how to build a term in memory. For this purpose, we consider an extension of the signature definition formalism: the make construct.

```
%op Nat zero {
  is_fsym(t) { t.getName().equals("zero") }
  make       { factory.makeAppl(factory.makeAFun("zero",0,false)) }
}

%op Nat suc(pred:Nat) {
  is_fsym(t)       { t.getName().equals("suc") }
  get_slot(pred,t) { (ATermAppl)t.getArgument(0) }
  make(t)          { factory.makeAppl(factory.makeAFun("suc",1,false),t) }
}
```

The makeAFun function has three arguments: the function name, the number of arguments and a boolean that indicates if the quotes are included in the function name or not.

Given this mapping, Tom can be used as previously: the following function implements the addition of two Peano integers:

```
public ATermAppl plus(ATermAppl t1, ATermAppl t2) {
  %match(Nat t1, Nat t2) {
    x,zero() -> { return ‘x; }
    x,suc(y) -> { return ‘suc(plus(x,y)); }
  }
  return null;
}
```

Please, note the use of ATermAppl in Java code, and the use of Nat in Tom construct.

3.1.3 Advanced examples

Let us suppose that we have a Java class Person with 2 getters (getFirstname and getLastname). In order to illustrate the signature definition formalism, we try to redefine (without using Vas) the abstract data type for the sort Person. The first thing to do consists in defining the Tom sort Person:

```
%typeterm Person {
  implement { Person }
}
```

To avoid any confusion, we use the same name twice: the Tom sort Person is implemented by the Java class Person. When declaring an operator, we defined the behavior as shown in the previous example:

```
%op Person person(firstname:String, lastname:String) {
  is_fsym(t) { t instanceof Person }
  get_slot(firstname,t) { t.getFirstname() }
  get_slot(lastname,t) { t.getLastname() }
  make(t0, t1) {  new Person(t0, t1) }
}
```

In this example, we illustrate another possibility offered by Tom: being able to know whether a term is rooted by a symbol without explicitly representing this symbol. The is_fsym(t) construct should return true when the term t is rooted by the algebraic operator we are currently defining. In this example, we say that the term t is rooted by the symbol person when the object t is implemented by an instance of the class Person. By doing so, we do not explicitly represent the symbol person, even if it could have been done via the reflective capabilities of Java (by using Person.getClass() for example).

3.1.4 Using list-matching

In this section, we show how to describe a mapping for associative operators.

```
%typeterm TomList {
  implement { ArrayList }
  equals(l1,l2)      { l1.equals(l2) }
}
```

Assuming that the sort Element is already defined, we can use the %oparray construct to define an associative operator. We also have to explain to Tom how to compute the size of a list, and how to access a given element. This is done via get_size and get_element constructs :

```
%oparray TomList conc( Element* ) {
  is_fsym(t)       { t instanceof ArrayList }
  get_element(l,n)   { (ATermAppl)l.get(n) }
  get_size(l)        { l.size() }
  make_empty(n)   { myEmpty(n) }
  make_append(e,l) { myAdd(e,(ArrayList)l) }
}
```

This construct is similar to %op except that additional informations have to be given: how to build an empty list (make_empty), and how to add an element to a given list (make_append). The auxiliary Java functions are defined as follows:

```
private ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}

private ArrayList myEmpty(int n) {
  ArrayList res = new ArrayList(n);
  return res;
}
```

Usually, we use an associative operator to represent (in a abstract way) a list data structure. There are many ways to implement a list, but the two most well-known are the use of array based list, and the use of linked-list. The previously described mapping shows how to map an abstract list to an array based implementation. Tom offers another similar constructs %oplist to map an associative operator to a linked-list based implementation.

Part II

Language

Chapter 4

Tom

4.1 The core language

Writing a good documentation is a difficult task. To help new users, as well as confirmed Tom developers, we have split the documentation into four complementary documents:

- the *reference manual* is a document which describes the language constructs, gives their precise syntax and informal semantics. To support the intuition, some examples may be given, but it is by no means a tutorial introduction to the language.
- the *user guide* describes how to use the Tom system in a production environment. This document explains both, how to install the software, and how to use it. In particular, the user guide describes the runtime library and external tools, like *vas*, which facilitates the use of Tom.
- the *tutorial* introduces, using examples, many concepts related to Tom, from simple to complex ones. The goal is to illustrate concepts and ideas introduced by the reference and the user manual.
- the *cookbook* is a collection of recipes that we use, and can be adapted to solve frequently encountered problems.

This document is intended as a reference manual for the Tom language, and thus precisely describes the constructs and their semantic.

4.2 Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (‘`like this`’). Non-terminal symbols are set in italic font (*like that*). Square brackets [...] denote optional components. Parentheses with a trailing star sign (...)* denotes zero, one or several repetitions of the enclosed components. Parentheses with a trailing plus sign (...)+ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

4.2.1 Lexical conventions

Identifier ::= *Letter* (*Letter* | *Digit* | ‘`_`’ | ‘`-`’)*
Integer ::= (*Digit*)+
Double ::= (*Digit*)+ [‘`.`’] (*Digit*)* | ‘`.`’ (*Digit*)+
String ::= ‘`"`’ (*Letter* | (‘\’ (‘`n`’ | ‘`t`’ | ‘`b`’ | ‘`r`’ | ‘`f`’ | ‘\’ | ‘`'`’ | ‘`"`’)))* ‘`"`’
Letter ::= ‘`A`’ ... ‘`Z`’ | ‘`a`’ ... ‘`z`’
Digit ::= ‘`0`’ ... ‘`9`’
Char ::= ‘`'`’ (*Letter* |*Digit*) ‘`'`’

4.2.2 Names

SubjectName	::=	*Identifier*
Type	::=	*Identifier*
SlotName	::=	*Identifier*
HeadSymbol	::=	*Identifier*
	|	*Integer*
	|	*Double*
	|	*String*
	|	*Char*
VariableName	::=	*Identifier*
AnnotedName	::=	*Identifier*
LabelName	::=	*Identifier*
FileName	::=	*Identifier*
AttributeName	::=	*Identifier*
XMLName	::=	*Identifier*
Name	::=	*Identifier*

4.3 Tom constructs

A Tom program is a host language program (namely C, Java, or Caml) extended by several new constructs such as %match, %strategy, %rule, %include, %vas, or backquote. Tom is a multi-languages compiler, so, its syntax depends on the host language syntax. But for simplicity, we only present the syntax of its constructs and explain how they can be integrated into the host language.

Using Java as *host-language*, the following Tom program is correct:

```
public class HelloWorld {
  %include { string.tom }

  public String getWord(String t) {
    %match(String t) {
      "World"  -> { return "World";}
      _        -> { return "Unknown"; }
    }
  }

  public final static void main(String[] args) {
    HelloWorld o = new HelloWorld();
    System.out.println("Hello " + o.getWord("World"));
  }
}
```

4.3.1 Tom program

Basically, a Tom program is list of blocks, where each block is either a Tom construct, or a sequence of characters. The idea is that that after transformation, the sequence of characters merged with the compiled Tom constructs should be a valid host language program. In the previous example, %include and %match constructs are replaced by function definitions and Java instructions, making the resulting program a correct Java program.

Syntax of a Tom program:

Tom ::= *BlockList*
BlockList ::=
(*MatchConstruct*
| *StrategyConstruct*
| *RuleConstruct*
| *BackQuoteTerm*
| *IncludeConstruct*
| *VasConstruct*
| *TypeTerm*
| *Operator*
| *OperatorList*
| *OperatorArray*
| '{' *BlockList* '}'
)*

- *MatchConstruct* is translated into a list of instructions. This construct may appear anywhere a list of instructions is valid in the host language.
- *StrategyConstruct* is translated into a class definition. Since it is translated into a class, this construct is valid only for Java.
- *RuleConstruct* is translated into a function definition. This construct may appear anywhere a function declaration is valid in the host language.
- *BackQuoteTerm* is translated into a function call.
- *IncludeConstruct* is replaced by the content of the file referenced by the construct. Tom looks for include files in : `./moduleName/`, `$TOM_HOME/share/jtom/` and `<path>`, where `<path>` is specified by option : `--import <path>`. If the file contains some Tom constructs, they are expanded.
- *VasConstruct* allows to define a VAS grammar. This construct is replaced by the content of the generated mapping. See the *user guide* for more details.
- *TypeTerm*, as well as *Operator*, *OperatorList*, and *OperatorArray* are replaced by some functions definitions.

4.3.2 Match construct

The %match construct (*MatchConstruct*) is one of the main contributions of Tom. This construct can be seen as an extension of the *SwitchCase* construct in C or Java, except that patterns are no longer restricted to constants (chars or integers). Given an object (the subject) and a list of patterns, our goal is to find the first pattern that *match* the subjects (i.e. that have a *compatible shape*). More formally, a pattern is a term built over variables and constructors. The latter ones describe the *shape* of the pattern, whereas the former ones are *holes* that can be instantiated to capture a value. When we consider the term $f(a, g(b))$, this has to be viewed as a tree based data-structure where a is the first child of the root, labeled by f. Similarly, b is the unique child of g, which is the second child of the root. We say that the pattern $f(x, y)$ matched this term (called subject), because we can give values to x and y such that the pattern and the subject become equal: we just have to assign a to x and $g(b)$ to y. Finding this assignment is called matching and instantiating. This is exactly what Tom is supposed to do. A pattern may of course contains subterms. Therefore, $f(x, g(b))$ or $f(a, g(y))$ are valid patterns which match against the subject.

Assuming that s is a Java variable of sort T, which references a term (the tree based object $f(a, g(b))$ for example), the following Tom construct is valid:

```
%match(T s) {
  f(a,g(y)) -> { /* code that uses y */ }
  f(x,g(b)) -> { /* code that uses x */ }
  f(x,y)    -> { /* code that uses x and y */ }
}
```

A *MatchConstruct* is composed of two parts:

- a list of host language variables (called *subjects*). These variables should reference the objects to be matched.
- a list of *PatternAction*: this is a list of pairs (pattern,action), where an action is a set of host language instructions which is executed each time a pattern matches the subjects.

The construct is defined as follow:

MatchConstruct ::= '%match' '(' *MatchArguments* ')' '{' (*PatternAction*)* '}'
MatchArguments ::= *Type SubjectName* (',' *Type SubjectName*)*
PatternAction ::= [*LabelName*':'] *TermList* ('|' *TermList*)* '->' '{' *BlockList* '}'
TermList ::= *Term* (',' *Term*)*

For expository reasons, we consider that a %match construct is evaluated in the following way:

- given a list of host language variables (they corresponds to objects only composed of constructors, called ground terms), the execution control is transferred to the first *PatternAction* whose patterns match the list of ground terms.
- given such a *PatternAction*, the list of variables is instantiated and the associated semantic action is executed. The instantiated variable are bound in the underlying host-language, and thus can be used in the action part.
- if the execution control is transferred outside the %match instruction (by a goto, break or return for example), the matching process is finished. Otherwise, the execution control is transferred to the next *PatternAction* whose patterns match the list of ground terms.
- when there is no more *PatternAction* whose patterns match the list of subjects, the %match instruction is finished, and the execution control is transferred to the next instruction.

The semantics of a match constructs may remind the switch/case construct, however, a big difference exists. In Tom, a theory like associativity may be attached to a constructor. Thus, given a subject and a pattern, there may exist several ways to match the pattern against the subject. Informally, when considering the subject $conc(a, g(b), c, g(d))$, and the pattern $conc(_^*, g(x), _^*)$, there are two possible match for this problem: either $x = b$, either $x = d$. note that $_^*$ is a special *hole* which can capture any sublist of $conc(\ldots)$.

When taking this new possibility into account, the evaluation of a %match construct is a bit more complex:

- given a *PatternAction* whose patterns match the list of ground terms, the list of variables is instantiated and the associated semantic action is executed.
- if the execution control is not transferred outside the %match instruction, in addition to the previous explanations, if the considered matching theory may return several matches, for each match, the free variables are instantiated and the associated semantic action is executed. This means that a same action may be executed several times, but in a different context: i.e. the variable have different instantiations.
- when all matches have been computed (there is at most one match in the syntactic theory), the execution control is transferred to the next *PatternAction* whose patterns match the list of ground terms.
- as before, when there is no more *PatternAction* whose patterns match the list of subject, the %match instruction is finished, and the execution control is transferred to the next instruction.

As mentioned in the BNF-syntax, a *label* may be attached to a pattern. In that case, in C and Java, it becomes possible to exit from the current *PatternAction* (using a goto or a break), without exiting from the whole %match construct. The control is transferred to the next pattern which matches the subjects. This feature is useful to exit from a complex associative-matching pattern which, for any reason, is no longer interesting.

Note: the behavior is not determined if a semantic action modifies a host language variable which is an argument of a %match instruction under evaluation.
Note: the ''|'' symbol can be used to introduce disjunction of pattern. This is a shortcut which avoid duplicating a *PatternAction*. However, this construct is **deprecated**. We recommend to use the *disjunction of symbols* construct, presented in the next section.

4.3.3 Tom pattern

As we can imagine, the behavior of a %match construct strongly depends on the patterns which are involved. The formalism which defines the syntax of a pattern is also an essential component of Tom. Unfortunately, its formal definition is complex, simply because there exist several ways to define patterns which have equivalent behaviors. On the other hand, the different shortcuts may help the programmer to simplify the definitions of patterns.

Informally, a pattern is a term that can be either a variable or an anonymous variable (x or _ for example). A pattern can also be composed of constructors (a, f(a), g(a,x), or h(a,_,x) for example). When a pattern matches a subject, it may be useful to keep a reference to a matched subterm. The annotation mechanism (z@g(y) for example) can be used for this purpose. Thus, considering the pattern $f(x, z@g(y))$ and the subject $f(a, g(h(b)))$, y is instantiated by $h(b)$, and z is instantiated by $g(h(b))$. This can be useful in C, to free the memory for example.

When identical actions have to be performed for a set of patterns which share a common structure, the *disjunction of symbols* may be used: pattern $(f \quad | \, g)(a)$ is equivalent to the set $\{f(a), g(a)\}$. The disjunction of symbols may also be used in subterm, like in $h((f \quad | \, g)(x))$.

More formally, a Tom term has the following syntax:

Term	::=	[*AnnotedName* '@'] *PlainTerm*
PlainTerm	::=	*VariableName* ['*']
	|	*HeadSymbolList* [*ExplicitTermList* | *ImplicitPairList*]
	|	*ExplicitTermList*
	|	'_'
	|	'_*'
	|	*XMLTerm*
HeadSymbolList	::=	*HeadSymbol*
	|	'(' *HeadSymbol* ('|' *HeadSymbol*)+ ')'
ExplicitTermList	::=	'(' *Term* (',' *Term*)* ')'
ImplicitPairList	::=	'[' *PairTerm* (',' *PairTerm*)* ']'
PairTerm	::=	'[' *SlotName* '=' *Term* ']'

A pattern is a term which could contain variables. When matching a pattern against a subject (a ground term), these variables are instantiated by the matching procedure (generated by Tom). In Tom, the variables do not have to be declared: their type is inferred automatically, depending on the context in which they appear.

As described previously, Tom offers several mechanisms to simplify the definition of a pattern:

- standard notation: a pattern can be defined using a classical prefix term notation. To make a distinction between variables and constants, it is recommended to explicitly write the empty list of arguments: for example, x(), denotes the constant x. In this case, the corresponding Tom operator (%op x) should have been declared. For simplicity, it is also possible to use the notation x, but note that the status of x depends on the existence of a Tom operator or not: x is a constant if %op x is defined, otherwise it is a variable.

- unnamed variable: the _ notation denotes an anonymous variable. It can be used everywhere a variable name can be used. It is useful when the instance of the variable does not need to be used. Similarly, the _* notation can be used to denote an anonymous list-variable. This last notation can improve the efficiency of list-matching because the instances of anonymous list-variables do not need to be built.

- annoted variable: the @ operator allows to give a variable name to a subterm. In f(x@g(_)) for example, x is a variable that will be instantiated by the instance of the subterm g(_). The variable x can then be used as any other variable.

- implicit notation: as explained below, the %op operator allows to give name to arguments. Assuming that the operator f has two arguments, named arg1 and arg2, then we can write the pattern f[arg1=a()] which is equivalent to f(a(),_). This notation can be interesting when using constructors with many subterms.

- unnamed list operator: it is often the case that given a list-sort, only one list-operator is defined. It this case, when there is no ambiguity, the name of the operator can be omitted. Considering the conc list-operator for example (see %oplist and %oparray below), to improve the readability, the pattern conc(_*,x,_*) can be written (_*,x,_*). This feature is particularly useful in the XML notation introduced in the following.

- symbol disjunction notation: to factorize the definition of pattern which have common subterms, it is possible to describe a family of patterns using a disjunction of symbols. The pattern (f|g)(a,b) corresponds to the disjunction f(a,b) or g(a,b). To be allowed in a disjunction (in standard notation), the constructors should have the same signature (arity, domain and codomain).

  In practice, it is usually better to use the disjunction notation with the implicit notation: ((f|g)[arg1=a()]). In that case, the signatures of symbols do not have to be identical: only involved slots have to be common (same names and types). Thus, the pattern (f|g)[arg1=a()] is correct, even if g has more slots than f: it only has to have the slot arg1, with the same sort.

  Note that the disjunction of symbol can also be used in XML notation: ¡a—b¿...¡/a—b¿.

4.3.4 XML pattern

To deal with XML documents, the XML notation can be used (<A><B attribute="name"/ ></A> for example).

When manipulating XML documents, we distinguish two main kinds of operations: retrieving information and transforming a document. Tom provides three different XML notations that ought to simplify the definition of patterns: the "standard " and the "implicit" XML notations are used to define compact (but incomplete) patterns. This notation is well suited to retrieve information. The "explicit" XML notation is used to precisely describe an XML pattern and all the variables that have to be instantiated. This notation is particularly well suited to perform XML transformation since it allows the programmer to precisely describe how variables have to be instantiated.

To make the XML notation understandable, we have to explain how XML documents are handled by Tom. To each XML document corresponds a DOM (Document Object Model) representation. In Tom, we have defined a mapping from DOM sorts to abstract algebraic sorts: TNode and TNodeList, which correspond respectively to Node and NodeList, defined by the Java DOM implementation.

Thus, a Node object becomes a ternary operator Element whose first subterm is the name of the XML node, the second subterm is a list of attributes and the third subterm is a list of subterms (which correspond to XML sub-elements). The second and the third elements are terms of sort TNodeList (because they are implemented by NodeList objects in DOM).

Thus, when considering the <A></A> XML document, the corresponding algebraic term is Element("A",[],[]), where [] denotes the empty list. Similarly, <A><Battribute="name"></B></A> is encoded into Element("A",[],[Element("B",[Attribute("attribute","name")],[])]).

When defining an XML pattern, the user has to introduce extra list-variables to precisely describe the XML pattern and capture the different contexts. Suppose that we are interested in finding a node <B></B> which is a subterm of a node <A></A> (but not necessary the first subterm). The algebraic pattern should be Element("A",[_*],[_*,Element("B",[_*],[_*]),_*]). Using the XML notation, this pattern should be expressed as follows: <A(_*)>(_*,<B(_*)>(_*)</B>,_*)</A>. This notation (called explicit) is precise but error prone. This is why we have introduced the explicit notation, where all context variable can be removed (and () are replaced by []): <A[]><B[]>[]</B></A>. The last notation (called standard XML notation) allows the user to remove the [] and replace the list-separator (,) by spaces. The previous pattern can be written: <A><B></B></A>.

These three different notations allow the user to choose the level of control he wants to have on the XML pattern matching algorithm.

The formal description of the syntax is the following:

```
XMLTerm          ::= '<' XMLNameList XMLAttributeList '/>'
                   | '<' XMLNameList XMLAttributeList '>' XMLChilds '</' XMLNameList '>'
                   | '#TEXT' '(' Identifier | String ')'
                   | '#COMMENT' '(' Identifier | String ')'
                   | '#PROCESSING-INSTRUCTION' '(' (Identifier | String) ',' (Identifier | String) ')'
XMLNameList      ::= XMLName
                   | '(' XMLName ( '|' XMLName )* ')'
XMLAttributeList ::= '[' [ XMLAttribute (',' XMLAttribute)* ] ']'
                   | '(' [ XMLAttribute (',' XMLAttribute)* ] ')'
                   | ( XMLAttribute )*
XMLAttribute     ::= '_*'
                   | VariableName '*'
                   | AttributeName '=' [AnnotedName '@'] ( Identifier | String )
                   | [AnnotedName '@'] '_' '=' [AnnotedName '@'] ( Identifier | String )
XMLChilds        ::= ( Term )*
                   | '[' Term ( ',' Term )* ']'
```

4.3.5 Rule construct

In Tom, we can also define a set of rewrite rules. All the left-hand sides should begin with the same root symbol:

```
RuleConstruct ::= '%rule' '{' ( Rule )* '}'
Rule          ::= RuleBody ( RuleCondition )*
RuleBody      ::= Term '->' Term
RuleCondition ::= 'where' Term ':=' Term
                | 'if' Term '==' Term
```

The %rule construct is composed of a list of conditional rewrite rules (the left-hand side is a term and the right-hand side is a term). All these rules (enclosed into a %rule { ... } construct) should begin with the same root symbol. The Tom compiler will generate a function (with a number of arguments equals to the arity of this root symbol) whose name corresponds to the name of this unique root symbol. Given a ground term, applying this function returns the instantiated and normalized right-hand side of the first rule (from top to bottom) whose pattern matches the considered subject and whose conditions are satisfied. When no rule can be applied (i.e. no pattern matches the subject, or no condition is satisfied), the given ground term, rooted by the root symbol of the rewrite system is returned.

In Tom, we consider two kinds of conditions:

- an equality condition ($t_1 == t_2$) is a pair of ground terms that belong to the same type. The condition is satisfied when the normal forms of the two terms (t_1 and t_2) are equal modulo the equals predicate defined in the definition of the type associated to t_1 and t_2.
- a matching condition ($p := t$) is a pair of terms where p may contain free variables. The condition is satisfied if the pattern p can be matched against the normal form of t. In this case, the free variables of p are instantiated and can be used in other conditions or the right-hand side of the rule.

4.3.6 Backquote construct

Another construct of Tom is the backquote (‘).

This construct can be used to build an algebraic term or to retrieve the value of a Tom variable (a variable instantiated by pattern-matching). The syntax of this operator is not fixed since it depends on the underlying language.

However, a backquote term should be of the following form:

- ‘*Name*: to denote a Tom variable
- ‘*Name*‘*’: to denote a Tom list-variable

- ‘*Name*(...): to build a prefix term
- ‘(...): to build an expression
- ‘*xml*(...): to build an XML term

In general, it is sufficient to add a backquote before the term you want to build to have the wanted behavior. The execution of ‘f(g(a)) will build the term $f(g(a))$, assuming that f, g, and a are Tom operators. Suppose now that g is no longer a constructor but a function of the host-language. The construction ‘f(g(a)) is still valid, but the semantic is the following: the constant a is built, then the function g is called, and the returned is put under the constructor f. Therefore, the result of g must be a correct term, which belongs to the right type (i.e. the domain of f).

To simplify the interaction with the host-language, it is also possible to use “unknown symbols” like f(x.g() or f(1+x). The scope of the backquote construct is determined by the scope of the most external enclosing braces, except in two case: ‘x and ‘x* which allows you to use variables instantiated by the pattern part. In that case the scope is limited to the length of the variable name, eventually extended by the ‘’*’’. Sometime, when writing complex expression like if(‘x==‘y —— ‘x==‘z), it can be useful to introduce extra braces (if(‘(x==y —— x==z))) to extend the scope of the backquote.

4.4 Tom signature constructs

4.4.1 Sort definition constructs

To define the mapping between the algebraic constructors and their concrete implementation, Tom provides a signature-mapping mechanism composed of several constructs. In addition to predefined mapping for usual builtin sorts (int, long, double, boolean, string, and char), all other algebraic sorts have to be declared using the %typeterm construct.

To use predefined sorts (in a %match construct or in the definition of a new operator), it is sufficient to use the %include construct (%include{ int.tom } for example).

When defining a new type with the %typeterm construct, two extra informations have to be provided:

- the implement construct describes how the new type is implemented. The host language part written between braces (’{’ and ’}’) is never parsed. It is used by the compiler to declare some functions and variables.
- the equals(t1,t2) construct corresponds to a predicate (parametrized by two term variables). This predicate should return true if the terms are “equal”. The true value should correspond to the builtin `true` value of the considered host language. This last optional predicate is used to compare builtin values and to compile non-linear left-hand sides.

Given a Java class Person we can define an algebraic mapping for this class:

```
%typeterm TomPerson {
  implement { Person }
  equals(t1,t2) { t1.equals(t2) }
}
```

Here, we assume that the method equals implement a comparison function over instances of Person. Note also that we have used TomPerson to make a clear distinction between algebraic sorts (defined in Tom) and implementation sorts (defined in Java, via the use of classes). In practice, we usually use the same name to denote both the algebraic sort and the implementation sort.

The grammar is the following:

IncludeConstruct ::= '%include' '{' *FileName* '}'
VasConstruct ::= '%vas' '{' *VasGrammar* '}'
GoalLanguageBlock ::= '{' *BlockList* '}'
TypeTerm ::= '%typeterm' *Type* '{'
KeywordImplement [*KeywordEquals*]
'}'
KeywordImplement ::= 'implement' *GoalLanguageBlock*
KeywordVisitorFwd ::= 'visitor_fwd' *GoalLanguageVisitorFwd*
KeywordEquals ::= 'equals' '(' *Name* ',' *Name* ')' *GoalLanguageBlock*

4.4.2 Constructor definition constructs

Once algebraic sorts are declared, Tom provides a mechanism to define sorted signatures for constructors using %op, %oplist or %oparray constructs. When defining a new symbol with the %op construct, the user should specify the name of the operator, its codomain, and its domain. The later one is defined by a list of pairs (slot-name, sort).

Let us consider again the class Person, and let us suppose that an instance of Person has two fields (name and age), we can define the following operator:

```
%op TomPerson person(name:String, age:int)
```

In this example, the algebraic operator person has two slots (name and age) respectively of sorts String and int, where String and int are pre-defined sorts.

In addition to the signature of an operator, several auxiliary functions have to be defined:

- The is_fsym(t) { predicate(t) } construct is used to check if a term t is rooted by the considered symbol. The `true` value should correspond to the builtin `true` value of the considered host language (`true` in Java or Caml, and something different from `0` in C for example).
- The make(t1,...,tn) construct is parametrized by several variables (i.e. that should correspond to the arity of the symbol). A call to this make function should return a term rooted by the considered symbol, where each subterm correspond to the terms given in arguments to the function. When defining a constant (i.e. an operator without argument, make can be defined without brace: make { ... }).
- The get_slot(slotName,t) construct has to be defined for all slots of the signature. The implementation of these constructs should be such that the corresponding subterm is returned.

Coming back to our example, checking if an object t is rooted by the symbol person can be done by checking that t is an instance of the class Person. Building a person can be done via the Java function new Person(...). Accessing to the slots name and age could be implemented by an access to the variables of the class Person. In practice, the following operator definition should work fine:

```
%op TomPerson person(name:String, age:int) {
  is_fsym(t)  { t instanceof Person }
  make(t1,t2) { new Person(t1,t2) }
  get_slot(name,t) { t.name } // assuming that 'name' is public
  get_slot(age,t)  { t.age  } // assuming that 'age' is public
}
```

When defining a new symbol with the %oplist construct, the user has to specify how the symbol is implemented. In addition, the user has to specify how a list can be built and accessed:

- the make_empty() construct should return an empty list.
- the make_insert(e,l) construct corresponds to a function parametrized by a list variable and a term variable. This function should return a new list `l'` where the element `e` has been inserted at the head of the list `l` (i.e. equals(get_head(l'),e) and equals(get_tail(l'),l) should be `true`).
- the get_head(l) function is parametrized by a list variable and should return the first element of the considered list.

- the get_tail(l) function is parametrized by a list variable and should return the tail of the considered list.

- the is_empty(l) constructs corresponds to a predicate parametrized by a list variable. This predicate should return `true` if the considered list contains no element.

Similarly, when defining a new symbol with the %oparray construct, the user has to specify how the symbol is implemented, how an array can be built, and accessed:

- the make_empty(n) construct should return a list of size `n`.

- the make_append(e,l) construct corresponds to a function parametrized by a list variable and a term variable.

  Warning: This function should return a list `l'` such that the element `e` is at the `n-th` position.

- the get_element(l,n) construct is parametrized by a list variable and an integer. This should correspond to a function that return the `n-th` element of the considered list `l`.

- the get_size(l) constructs corresponds to a function that returns the size of the considered list. By convention, an empty list contains `0` element.

The %oplist or %oparray is complex but not difficult to use. Let us consider the ArrayList Java class, and let us define a Tom mapping over this data-structure. The first thing to do consists in defining the sort for the elements and the sort for the list-structure:

```
%typeterm Object {
  implement     { Object        }
  equals(l1,l2) { l1.equals(l2) }
}
%typeterm TomList {
  implement     { ArrayList    }
  equals(l1,l2) { l1.equals(l2) }
}
```

Once defined the sorts, it becomes possible to define the list-operator TomList conc(Object*). This operator has a variadic arity: it takes several Object and returns a TomList.

```
%oparray TomList conc( Object* ) {
  is_fsym(t)       { t instanceof ArrayList }
  make_empty(n)    { new ArrayList(n)       }
  make_append(e,l) { myAdd(e,(ArrayList)l)   }
  get_element(l,n) { (Object)l.get(n)        }
  get_size(l)      { l.size()                }
}

private ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}
```

An auxiliary function myAdd is used since the make_append construct should return a new list. The get_element should return an element whose sort belongs to the domain (Object) in this example. Although not needed in this example, in general, a cast ((Object)l.get(n)) is needed.

The grammar for the mapping constructs is the following:

Operator ::= '%op' *Type Name* ['(' *SlotName* ':' *Type* (',' *SlotName* ':' *Type*)* ')']
'{' *KeywordIsFsym* (*KeywordMake* | *KeywordGetSlot*)* '}'

OperatorList ::= '%oplist' *Type Name* '(' *Type* '*' ')'
'{' *KeywordIsFsym* (*KeywordMakeEmptyList* | *KeywordMakeInsert* | *KeywordGetHead* | *KeywordGetTail* | *KeywordIsEmpty*)* '}'

OperatorArray ::= '%oparray' *Type Name* '(' *Type* '*' ')'
'{' *KeywordIsFsym* (*KeywordMakeEmptyArray* | *KeywordMakeAppend* | *KeywordElement* | *KeywordGetSize*)* '}'

KeywordIsFsym ::= 'is_fsym' '(' *Name* ')' *GoalLanguageBlock*

KeywordGetSlot ::= 'get_slot' '(' *Name* ',' *Name* ')' *GoalLanguageBlock*

KeywordMake ::= 'make' ['(' *Name* (',' *Name*)* ')'] *GoalLanguageBlock*

KeywordGetHead ::= 'get_head' '(' *Name* ')' *GoalLanguageBlock*

KeywordGetTail ::= 'get_tail' '(' *Name* ')' *GoalLanguageBlock*

KeywordIsEmpty ::= 'is_empty' '(' *Name* ')' *GoalLanguageBlock*

KeywordMakeEmptyList ::= 'make_empty' ['(' ')'] *GoalLanguageBlock*

KeywordMakeInsert ::= 'make_insert' '(' *Name* ',' *Name* ')' *GoalLanguageBlock*

KeywordGetElement ::= 'get_element' '(' *Name* ',' *Name* ')' *GoalLanguageBlock*

KeywordGetSize ::= 'get_size' '(' *Name* ')' *GoalLanguageBlock*

KeywordMakeEmptyArray ::= 'make_empty' '(' *Name* ')' *GoalLanguageBlock*

KeywordMakeAppend ::= 'make_append' '(' *Name* ',' *Name* ')' *GoalLanguageBlock*

4.4.3 Predefined sorts and operators

As mentioned previously, the system comes with several predefined signature-mappings for C, Java and Caml. Among them, let us mention:

- int.tom (written 1, 2, 3, *etc.*)
- long.tom (written 1l or 1L, *etc.*)
- double.tom (using Java grammar for double)
- boolean.tom (true or false)
- char.tom (written 'a', 'b', *etc.*)
- string.tom (written "a", "ab", *etc.*)

These mappings define, for each builtin sort of the host language, an algebraic sort that can be used in a %match or a signature definition construct. Thus, builtin values, such as f(5), g('a') or h("foo") , can be use in patterns.

The string.tom mapping is interesting because it provides an associative operator (concString) which allows the programmer to consider a string as a list of characters. Thus, the string "foo" can be seen as the algebraic object concString('f','o','o'). By using this mapping, it becomes possible to perform pattern matching against the content of a string. The pattern concString('f',X*) will match any string which begins with the character 'f'. By using the unnamed-symbol capability, this pattern can be written: ('f',X*) instead of concString('f',X*).

To match any string which begins with the substring "fo", the corresponding pattern should be ('f','o',X*). To simplify the definitions of such patterns, Tom supports an exception which allows the programmer to write ('fo',X*) instead. Internally, the ill-formed character 'fo' is expanded into the list ('f','o'). Thus, the annotated notation y@'fo' is expanded into (y@'f',y@'o'), and thus may be dangerous to use in this case.

In addition to these mappings, several other predefined mappings come with the system:

- aterm.tom and atermlist.tom provide a mapping to the C and Java version of the ATerm Library
- list.tom provides a mapping to the builtin notion of List in Caml
- dom.tom provides a mapping to the Java version of the DOM Library

Chapter 5

Gom

Gom is a generator of tree implementations in Java, allowing the definition of a preferred canonical form, using *hooks*. The tree implementations Gom generates are characterized by strong typing, immutability (there is no way to manipulate them with side-effects), maximal subterm sharing and the ability to be used with strategies.

5.1 Gom syntax

The basic functionality of Gom is to provide a tree implementation in Java corresponding to an algebraic specification.

Gom provide a syntax to concisely define abstract syntax tree, which is inspired from the algebraic type definition of ML languages.

Each Gom file consist in the definition of one or more modules. In each module we define sorts, and operators for the sorts. Additionally, it allows to define *hooks* modifying the default behavior of an operator.

Module	::=	'`module`' *ModuleName* [*Imports*] *Grammar*
Imports	::=	'`imports`' (*ModuleName*)*
Grammar	::=	'`abstract syntax`' (*TypeDefinition* |*HookDefinition*)*
TypeDefinition	::=	*SortName* '`=`' ['`|`'] *OperatorDef* ('`|`' *OperatorDef*)*
OperatorDef	::=	*OperatorName* '`(`'[*SlotDef*('`,`'*SlotDef*)*]'`)`'
	|	*OperatorName* '`(`' *SortName*'`*`' '`)`'
SlotDef	::=	*SlotName* '`:`' *SortName*
HookDefinition	::=	*OperatorName* '`:`' *HookOperation* '`{`' *TomCode* '`}`'
HookOperation	::=	('`make`' |'`make_insert`') '`(`' (*Identifier*)* '`)`'
ModuleName	::=	*Identifier*
SortName	::=	*Identifier*
OperatorName	::=	*Identifier*
SlotName	::=	*Identifier*

5.1.1 Builtin sorts and operators

Gom supports integers and String builtins. Those builtins define respectively the sorts int and String, that can be used as sorts for slot definitions. It is not possible to define a new operator those domain is one of the builtin sorts, since those sorts are defined in another module. Also, to be able to use one or the other builtin sorts, it is necessary to declare the import of the corresponding module in the *Imports* section (those modules being called int and String).

5.1.2 Example of signature

The syntax of Gom is quite simple and can be used to define many-sorted abstract-datatypes. The module section defines the name of the signature. The imports section defines the name of the imported

signatures. The abstract syntax part defines the operators with their signature. For each argument, a sort and a name (called slot-name) has to be given.

```
module Expressions
imports String int

abstract syntax
Bool = True()
     | False()
     | Eq(lhs:Expr, rhs:Expr)
Expr = Id(stringValue:String)
     | Nat(intValue:int)
     | Add(lhs:Expr, rhs:Expr)
     | Mul(lhs:Expr, rhs:Expr)
```

The definition of signature in Gom have several restrictions:

- there is no overloading: two operators cannot have the same name
- for a given operator, all slot-names must be different
- if two slots have the same slot-name, they must belong to the same sort. In the previous example, Eq, Add, and Mul can have a slot called lhs of sort Expr. But, Id and Nat cannot have a same slot named value, since their sort are not identical (the slots are respectively of sorts String and int for Id and Nat).

5.2 Generated API

For each *Module*, the Gom compiler generates an API specific of this module, possibly using the API of other modules (declared in the *Imports* section). This API is located in a `Java` package named using the *ModuleName* lowercased, and contains the tree implementation itself, with some additional utilities:

- An AbstractType class: the class "*ModuleName*AbstractType" is the generic type for all nodes those type is declared in this module. It declares generic functions for all tree nodes: a toATerm() method returning a aterm.ATerm representation of the tree; a `symbolName()` method returning a String representation for the function symbol of the tree root; the toString() method, which returns a string representation. It declares also all accept(Visitor v) to implement the visitor combinators pattern used by the strategy language.
- the Visitor, BasicStrategy and Forward classes for the module, respectively named *ModuleName*Visitor, *ModuleName*BasicStrategy, *ModuleName*Forward. Those classes implement the visitor combinator design pattern for the module, by providing a typed Visitor interface (one visit method per sort), a Forward visitor providing a simple forwarding visitor respecting types, and a BasicStrategy, which is a visitable visitor built on top of the Forward class.
- in a subpackage types, Gom generates one class for each sort defined in the module, those name is equal to the sort name. Each sort class extends AbstractType for the module, and declares methods `boolean is`*OperatorName*() for each operator in the module (false by default). It declares getters (methods named get*SlotName*) for each slot used in any operator of the sort (throwing an exception by default). A method SortName fromTerm(aterm.ATerm trm) is generated, allowing to use ATerm as an exchange format.
- for each sort, generate classes for the operators in a package types.*SortName*. All classes for the operator are named following the operator name, and extend the class generated for the corresponding sort. Thus they provide getters for the slots of the operator, and the is*OperatorName*() method is overriden to return true. Those classes all implement the jjtraveler.Visitable interface. It is worth noting that builtin fields are not accessible from the jjtraveler.Visitable, and thus will not be visited by strategies. The operator class also implements a static make method, building a new instance of the operator. This make method is the only way to obtain a new instance.

- for each module, one file *ModuleName*`.tom` providing a Tom mapping for the sorts and operators defined or imported by the module.

5.2.1 Hooks

Hooks are used to alter operations of the operators. '`make`' and '`make_insert`' hooks are altering the creation operations for respectively algebraic and variadic operators. '`make_insert`' is simply a derivative case of '`make`' for the Cons of variadic operators (see 5.2.3).

The hook operation type is followed by a list of arguments name between '`()`'. The creation operation takes those arguments in order to build a new instance of the operator. Thus, the arguments number has to match the slot number of the operator definition, and types are inferred from this definition.

Then the body of the hook definition is constructed of Java of Tom code. the Tom code is compiled using the mapping definition for the current module, and thus allows to build and match terms from the current module. This code can also use the realMake function, which consists in the "inner" default allocation function. This function takes the same number and types of arguments as the hook. In any case, if the hooks code do not perform a return itself, this realMake function is called with the hooks arguments at the end of the hook execution.

5.2.2 Hook example

Using the expression example introduced 5.1.2, we can add *hooks* to implement the computation of Add and Mul when both arguments are known integers (i.e. when they are Nat(x))

```
module Expressions
imports String int

abstract syntax
Bool = True()
     | False()
     | Eq(lhs:Expr, rhs:Expr)
Expr = Id(stringValue:String)
     | Nat(intValue:int)
     | Add(lhs:Expr, rhs:Expr)
     | Mul(lhs:Expr, rhs:Expr)
Add:make(l,r) {
  %match(Expr l, Expr r) {
    Nat(lvalue), Nat(rvalue) -> {
      return `Nat(lvalue + rvalue);
    }
  }
}
Mul:make(l,r) {
  %match(Expr l, Expr r) {
    Nat(lvalue), Nat(rvalue) -> {
      return `Nat(lvalue * rvalue);
    }
  }
}
```

Using this definition, it is impossible to have an expression containing unevaluated expressions where a value can be calculated. Thus, a procedure doing constant propagations for Id those value is know could simply replace the Id by the corresponding Nat, and rely on this mechanism to evaluate the expression. Note that the arguments of the make hook are themselves elements build on this signature, and thus the hooks have been applied for them. In the case of hooks encoding a rewrite system, this corresponds to using an innermost strategy.

5.2.3 Variadic operators

Variadic operators are operators that have no fixed arity. They can have any number of arguments, and are usually used to represent lists. Variadic operators are declared using the syntax:

OperatorDef ::= *OperatorName* '(' *SortName*'*' ')'

The generated code contains two operator classes: one `Empty`*Operator* which is used to represent the empty list, and one `Cons`*Operator* having two fields: one with the codomain sort, and one with the domain sort of the variadic operator. This allows to define lists as the composition of many `Cons` and one `Empty` objects.

As the interface is different than the one for standard algebraic operators, the `make_insert` hook operation is used to modify the creation operation for variadic operators.

5.3 Combining Gom with Tom

A first solution to combine Tom with Gom is to use Gom as a standalone tool, using the command line tool or the ant task.

In that case, the module name of the Gom specification and the package option determine where the files are generated. To make things correct, it is sufficient to import the generated Java classes, as well as the generated Tom file. In the case of a Gom module called Module, all files are generated in a directory named module and the Tom program should do the following:

```
import module.*;
import module.types.*;

class MyClass {
  ...
  %include { module/Module.tom }
```

A second possibility to combine Tom with Gom is to use the %gom construct offered by Tom. In that case, the Gom module can be directly included into the Tom file, using the %gom instruction:

```
package myPackage;

import myPackage.myclass.expressions.*;
import myPackage.myclass.expressions.types.*;

class MyClass {
  %gom{
    module Expressions

    abstract syntax
    Bool = True()
         | False()
      ...
    Expr = Mul(lhs:Expr, rhs:Expr)
  }
```

Note that the Java code is generated in a package that corresponds to the current package, followed by the class-name and the module-name. This allows to define the same module in several classes, and avoid name clashes.

Chapter 6

Strategies

Recently, we have added a new library, inspired by ELAN, Stratego, and JJTraveler, which allows to easily define various kinds of traversal strategies. mutraveler.tom provides an algebraic view of elementary strategy constructors.

6.1 Basic operators

The following operators are the key-component that can be used to define more complex strategies. In this framework, the application of a strategy to a term can fail. In Java, the *failure* is implemented by an exception (VisitFailure).

Identity@2)	=¿	t
Fail@2)	=¿	*failure*
Sequence(s1,s2)@2)	=¿	*failure* if *s1*@2) fails *s2*@2) if *s1*@2) =¿ t'
Choice(s1,s2)@2)	=¿	t' if *s1*@2) =¿ t' *s2*@2) if *s1*@2) fails
All(s)@2)	=¿	f(t1',...,tn') if *s1*@2) =¿ t1', ..., *sn*@2) =¿ tn' *failure* if there exists i such that *si*@2) fails
All(s)@2)	=¿	c
One(s)@2)	=¿	f(t1,...,ti',...,tn) if *si*@2) =¿ ti' *failure* *s1*@2) fails, ..., *sn*@2) fails
One(s)@2)	=¿	*failure*
Omega(i,s)@2)	=¿	f(t1,...,ti',...,tn) if *si*@2) =¿ ti' *failure* if *si*@2) fails

6.2 Basic operators with identity considered as failure

In order to get more efficient strategies (in particular when performing leftmost-innermost normalization), we consider variants where the notion of *failure* corresponds to the *identity*. This means that when a term cannot can be transformed by a strategy (into a different term), this is considered as a *failure*. In this context, several new basic operators are introduced.

SequenceId(s1,s2)@2)	=¿	*s2*@2) if *s1*@2) =¿ t' with t!=t t otherwise
ChoiceId(s1,s2)@2)	=¿	t' if *s1*@2) =¿ t' with t!=t' *s2*@2) otherwise
OneId(s)@2)	=¿	f(t1,...,ti',...,tn) if *si*@2) =¿ ti' with ti!=ti' f(t1,...,tn) otherwise
OneId(s)@2)	=¿	cst

6.3 Some basic strategies

In order to define recursive strategies, we introduce the *mu* abstractor. This allows to give a name to the current strategy, which can be referenced later.

Try(s)	=	Choice(s,Identity)
Repeat(s)	=	mu x.Choice(Sequence(s,x),Identity())
OnceBottomUp(s)	=	mu x.Choice(One(x),s))
BottomUp(s)	=	mu x.Sequence(All(x),s))
Innermost(s)	=	mu x.Sequence(All(x),Try(Sequence(s,x))))

The *Try* strategy never fails: it tries to apply the strategy *s*. If it succeeds, the result is returned. Otherwise,the *Identity* strategy is applied, and the subject is not modified.

The *Repeat* strategy applies the strategy *s* as many times as possible, until a failure occurs. The last unfailing result is returned.

The strategy *OnceBottomUp* tries to apply the strategy *s* once, starting from the leftmost-innermost leaves. *BottomUp* looks like *OnceBottomUp* but is not similar: *s* is applied to all nodes, starting from the leaves. Note that the application of *s* should not fail, otherwise the whole strategy also fails.

The strategy *Innermost* tries to apply *s* as many times as possible, starting from the leaves. This construct is useful to compute normal forms.

6.4 Some basic strategies with identity considered as failure

TryId(s)	=	s
RepeatId(s)	=	mu x.SequenceId(s,x))
OnceBottomUpId(s)	=	mu x.ChoiceId(OneId(x),s))
OnceTopDownId(s)	=	mu x.ChoiceId(s,OneId(x)))
InnermostId(s)	=	mu x.Sequence(All(x),SequenceId(s,x)))
OutermostId(s)	=	mu x.Sequence(SequenceId(s,x),All(x)))

6.5 Using MuTraveler in Tom

In order to use the runtime library and the JJTraveler framework, we need to import the following packages:

```
import tom.library.strategy.mutraveler.*;
import jjtraveler.reflective.VisitableVisitor;
import jjtraveler.Visitable;
import jjtraveler.VisitFailure;
```

We also need to import the corresponding mapping:

```
%include { mutraveler.tom }
```

Then we define an elementary strategy:

```
%strategy RewriteSystem() extends `Fail(){

  visit Term {
      a()        -> { return `b(); }
      b()        -> { return `c(); }
      g(c(),c()) -> { return `c(); }
  }
}
```

Then, it becomes quite easy to define various strategies on top of this elementary strategy:

```
Term subject = `f(g(g(a,b),g(a,a)));
VisitableVisitor rule = `RewriteSystem();
try {
  System.out.println("subject       = " + subject);
  System.out.println("onceBottomUp  = " + MuTraveler.init(`OnceBottomUp(rule)).visit(subject));
  System.out.println("onceBottomUpId= " + MuTraveler.init(`OnceBottomUpId(ruleId)).visit(subject));
  System.out.println("bottomUp      = " + MuTraveler.init(`BottomUp(Try(rule))).visit(subject));
  System.out.println("bottomUpId    = " + MuTraveler.init(`BottomUp(ruleId)).visit(subject));
  System.out.println("innermost     = " + MuTraveler.init(`Innermost(rule)).visit(subject));
  System.out.println("innermostSlow = " + MuTraveler.init(`Repeat(OnceBottomUp(rule))).visit(subject));
  System.out.println("innermostId   = " + MuTraveler.init(`InnermostId(ruleId)).visit(subject));
} catch (VisitFailure e) {
  System.out.println("reduction failed on: " + subject);
}
```

6.6 Strategy construct

In Tom, we can define an elementary strategy to use the MuTraveler library :

StrategyConstruct	::=	'%strategy' *StrategyName* '(' [*StrategyArguments*] ')' '`extends`' '`' *ExtendsTerm* '{' *StrategyVisitList* '}'
StrategyArguments	::=	*AlgebraicType SubjectName* (',' *AlgebraicType SubjectName*)*
StrategyVisitList	::=	(*StrategyVisit*)*
StrategyVisit	::=	'visit' *AlgebraicType* '{' (*PatternAction*)* '}'

An elementary strategy has a name, followed with mandatory parenthesis. Inside these parenthesis, we have optional parameters. These parameters must be algebraic types. If for instance you need to pass Java variables, you have to use mappings to do so. The strategy has an extends term which defines the behavior of the default strategy that will be applied.
The body of the strategy is a list of visited sorts. Each StrategyVisit contains a list of PatternAction that will be applied to the corresponding sort. In other words, a StrategyVisit is translated into a MatchStatement.
For strategies, all AlgebraicType needs a TomForwardType. This TomForwardType is used to know how sorts implements the interface VisitableVisitor. For a given strategy, all visited sorts must have the same TomForwardType.
Once you defined a strategy, you can instantiate it:

```
`StrategyName(StrategyAlgebraicVariables)
```

Chapter 7

Runtime Library

Predefined mappings

The system comes with several predefined signature-mappings for C, Java and Caml. Among them, let us mention:

- boolean.tom
- char.tom
- double.tom
- int.tom
- long.tom
- string.tom

These mappings define for each builtin sort of the host language an algebraic sort that can be used in a %match or a signature definition construct. Thus, builtin values, such as f(5), g('a') or h("foo"), can be used in patterns.

The string.tom mapping provides an associative operator (concString) which allows the programmer to consider a string as a list of characters. Thus, the string "foo" can be seen as the algebraic object concString('f','o','o'). By using this mapping, it becomes possible to perform pattern matching against the content of a string. The pattern concString('f',X*) will match any string which begins with the character 'f'. By using the unnamed-symbol capability, this pattern can be written: ('f',X*) instead of concString('f',X*).

To match any string which begins with the substring "fo", the corresponding pattern should be ('f','o',X*). To simplify the definitions of such patterns, Tom supports an exception which allows the programmer to write ('fo',X*) instead. Internally, the ill-formed character 'fo' is expanded into the list ('f','o'). Thus, the annotated notation y@'fo' is expanded into (y@'f',y@'o'), and thus may be dangerous to use in this case.

In addition to these mappings, several other predefined mappings come with the system:

- aterm.tom and atermlist.tom provide a mapping to the C and Java version of the ATerm Library
- util/ArrayList.tom and util/LinkedList.tom provide a mapping to the ArrayList and LinkedList implementation from the Java standard library
- list.tom provides a mapping to the builtin notion of List in Caml

XML

To support the transformation of XML documents, Tom provides a specific syntax for defining patterns, as well as several predefined mappings:

- dom.tom: maps XML notation to a Java implementation of the DOM library
- dom_1.5.tom: maps XML notation to a Java (version 1.5) implementation of the DOM library
- TNode.tom: maps XML notation to an ATerm based representation, generated by ApiGen

See Section 4.3.4 for a detailed description of the XML facilities offered by Tom.

Chapter 8

Migration Guide

This chapter documents important changes in the Tom language that impact running code. Thus, it exposes how Tom code for an older version should be changed to be able to compile and work under a newer version.

8.1 Migration from 2.2 to 2.3

- To make the migration from Vas to Gom easy, the Gom compiler do accept the Vas syntax. Since the Vas compatibility syntax should not be used in new Gom definitions, we will focus in this guide on the pure Gom syntax.

8.2 Migration from 2.1 to 2.2

Mapping definition

Since version 2.2, the mapping definition formalism has been simplified. Therefore, we are no longer able to be fully compatible with the old syntax. Some small and simple modifications have to be performed:

- in %op, the slot-names are no longer optional. For each argument of a constructor, a name has to be given. This name can then be used within the "bracket" notation.

  %op T f(T,T) has to be replaced by %op T f(arg1:T,arg2:T), where arg1 and arg2 are slot-names.

- for each slot-name, a get_slot(name,t) construct has to be specified. This defines, given an object t, how to access to the slot-name named name.

  The definition of get_slot can be defined by using the previously defined construct get_subterm (the first slot corresponds to the $0-th$ subterm, second slot to the $1-st$ subterm, *etc.*). Thus, for the previous example, we can have:

```
%op T f(arg1:T, arg2:T) {
  get_slot(arg1,t) { get_subterm(t,0) }
  get_slot(arg2,t) { get_subterm(t,1) }
}
```

- for each operator (%op, %oplist, and %oparray), the is_fsym(t) predicate has to be defined. This function should return true when t is rooted by the considered symbol, and false otherwise. In practice, the constructs can be deduced by combining get_fun_sym, cmp_fun_sym, and fsym:

```
%op T f(arg1:T, arg2:T) {
  is_fsym(t) { cmp_fun_sym(get_fun_sym(t), body of the old fsym construct) }
  get_slot(arg1,t) { get_subterm(t,0) }
  get_slot(arg2,t) { get_subterm(t,1) }
```

```
}
```

- in %oplist, get_head, get_tail, and is_empty have to be defined. In practice, these definitions have to be copied from %typelist to %oplist
- in %oparray, get_element, and get_size have to be defined. In practice, these definitions have to be copied from %typearray to %oparray
- %typelist, and %typearray do no longer exist: %typeterm has to be used to define any kind of sort. This is not a problem since access functions are now defined inside operator definitions (%op, %oplist, or %oparray)
- get_fun_sym, cmp_fun_sym, and get_subterm do no longer exist in %typeterm
- fsym do no longer exists in %op, %oplist, and %oparray

As an example, let us consider the following "old" mapping definition:

```
%typeterm TomTerm {
  implement          { ATermAppl        }
  cmp_fun_sym(t1,t2) { t1 == t2         }
  get_fun_sym(t)     { t.getName()      }
  get_subterm(t, n)  { t.getArgument(n) }
  equals(t1, t2)     { t1 == t2         }
}

%typelist TomList {
  implement          { ATermList                }
  get_fun_sym(t)     { ((t instanceof ATermList)?"conc":null) }
  cmp_fun_sym(t1,t2) { t1 == t2                 }
  equals(l1,l2)      { l1 == l2                 }
  get_head(l)        { (ATermAppl)l.getFirst() }
  get_tail(l)        { l.getNext()              }
  is_empty(l)        { l.isEmpty()              }
}

%op TomTerm a {
  fsym   { "a" }
  make() { factory.makeAppl(factory.makeAFun("a", 0, false)) }
}

%op TomTerm f(TomTerm) {
  fsym     { "f" }
  make(t1) { factory.makeAppl(factory.makeAFun("f", 1, false),t1) }
}

%oplist TomList conc( TomTerm* ) {
  fsym             { "conc"             }
  make_empty()     { factory.makeList() }
  make_insert(e,l) { l.insert(e)        }
}
```

The two first type definitions have to be replaced by:

```
%typeterm TomTerm {
  implement      { ATermAppl }
  equals(t1, t2) { t1 == t2  }
```

```
}

%typeterm TomList {
  implement     { ATermList }
  equals(l1,l2) { l1 == l2  }
}
```

The operator definition can be replaced by the following code:

```
%op TomTerm a {
  is_fsym(t) { t.getName() == "a" }
  make()     { factory.makeAppl(factory.makeAFun("a", 0, false)) }
}

%op TomTerm f(arg:TomTerm) {
  is_fsym(t)      { t.getName() == "f" }
  get_slot(arg,t) { (TomTerm) t.getArgument(t,0) }
  make(t1)        { factory.makeAppl(factory.makeAFun("f", 1, false),t1) }
}

%oplist TomList conc( TomTerm* ) {
  is_fsym(t)       { t instanceof ATermList }
  get_head(l)      { (ATermAppl)l.getFirst() }
  get_tail(l)      { l.getNext() }
  is_empty(l)      { l.isEmpty() }
  make_empty()     { factory.makeList() }
  make_insert(e,l) { l.insert(e) }
}
```

Disjunction of patterns

Since version 2.2, the disjunction of patterns construct has become deprecated. It can still be used, but we recommend to use the disjunction of symbols instead. This last construct allows to share patterns which have similar subterms but different head-symbols. Therefore, the construct g(g(a)) | g(h(a)) can be replaced by g((g | h)(a)). Since version 2.2, this construct has been extended to support disjunction of symbols which do not have identical signature: only involved slots have to be compatible.

Considering %op T f(arg1:T, arg2:T) and %op T g(arg1:T), it is now possible to write g((g | f)[arg1=a]).

Variable

To prepare a future extension where variables have to be distinguished from constructors of arity 0, we have turned into error all ambiguities between variables and constructors. As a consequence, a variable cannot any longer have the same name as a constant.

Vas

Since version 2.2, ApiGen and Vas have been updated:

- name of list-operators defined in Vas are now significant and can be used in match constructs.
- generated Tom mappings are now stored in the same directory as the generated factory. In practice, imports of the form %include{ file.tom } have to be replace by %include{ package/file.tom }.

Strategy library

The TravelerFactory class has been removed and replaced by a Tom mapping: mutraveler.tom. Therefore, backquote notation has to be used: travelerFactory.Repeat(travelerFactory.OnceBottomUp(rule)) is replaced by ‘Repeat(OnceBottomUp(rule)).

8.3 Migration from 2.0 to 2.1

Tom library

Since version 2.1, the runtime library has been reorganized. Java importations have to be updated according to the following hierarchy:

```
tom
 |--- library
 |        |--- adt               (datatype definitions)
 |        |--- mapping           (predefined mapping)
 |                |--- adt       (generated mapping)
 |        |--- plugin            (platform tools)
 |        |--- set               (to handle sets)
 |        |--- strategy
 |                |--- concurrent (to support parallel strategy)
 |        |--- traversal         (generic traversal)
 |        |--- xml               (xml tools)
 |
 |--- platform                   (the Tom Plugin Platform, known as Tom server)
          |--- adt
```

Command line options

- `--protected` is a new option which makes Tom generates protected access functions. By default, access functions are now private instead of public,
- `--noWarning` has been replaced by its dual `--wall` (for printing warnings).

8.4 Migration from 1.5 to 2.0

New backquote usage

Since version 2.0, Tom integrates an optimizer which removes unused variable assignments and performs inlining optimizations. In order to make these optimizations correct, the Tom compiler needs to know whether a Tom variable (instantiated by pattern matching) is used or not.

In previous versions of Tom, it was possible to use or re-assign a Tom variable without any particular requirement. Since version 2.0, it is recommended (and needed if you want to use the optimizer) to use the backquote mechanism to retrieve the value of a Tom variable. The syntax of a backquote construct has been modified in the following way:

- ‘*Name* (like ‘x) is now correct. This allows access to the value of a Tom variable. This construct also allows access to the value of list-variable, but the ‘*Name*‘* construct is to be preferred,
- ‘*Name*‘* (like ‘X*) is now correct. This allows access to the value of a Tom list-variable,
- ‘*Name*(...) is correct, as in previous versions. This construct builds a term in memory,
- ‘(...) is now correct and can be used to simulate the previous syntax. Since version 2.0, it is no longer possible to start a backquote construct by using something different from a *Name* or a ‘(’. Thus, ‘1+x is no longer valid: ‘(1+x) or 1+‘x has to be used instead,
- ‘*xml*(...) has to be used to create an XML term.

Command line options

- `--destdir` (short version: `-d`) option: as in `javac`, this option specify where to place generated files,

- `--output` (short version: `-o`) option: as in `gcc`, this option specify the name of the *unique* generated file. This option is not compatible with `--destdir`,
- `-o` is no longer the short version of `--noOutput`.

Predefined mapping

Since version 2.0, Tom does no longer support predefined builtin sorts, via %typeint, %typestring or %typedouble. Tom comes now with a library of predefined mappings. Among them, int.tom, string.tom and double.tom. In order to use these mappings, the %include construct should be used (i.e. %include{ int.tom } to import the predefined int mapping).

Part III

Tools

Chapter 9

Installation

This chapter describes how to install and use the Tom system.

9.1 Requirements

Tom is a Java application written with JDK 1.4. Tom is platform independent and runs on various systems. It has been used successfully on many platforms, including Linux (Mandrake, Debian), FreeBSD, NetBSD, MacOS X, and Windows XP.

The only requirement to run Tom is to have a recent Java Runtime Environment installed (version 1.4.x or newer) and/or the Eclipse platform (3.0 or newer). In addition, Ant can be useful to compile the system and the examples:

- Java is available for download at java.sun.com.
- Eclipse is available for download at Eclipse
- Ant is available for download at ant.apache.org

9.2 Windows

9.3 Unix

The binary distribution of Tom consists of the following directory layout:

```
jtom
 +--- bin  // contains launcher scripts
 |
 +--- doc  // contains documentation
 |
 +--- lib  // contains Tom jars and necessary dependencies
 |
 +--- share
       +--- contrib
       |
       +--- jtom    // contains predefined mapping
       |     +--- adt
       |     |
       |     +--- c
       |     |
       |     +--- caml
       |     |
       |     +--- java
```

```
|
+--- man
```

Only the bin/ and lib/ directories are required to run Tom. To install Tom, choose a directory and copy the distribution files there. This directory will be known as TOM_HOME.

Setup

Before you can run Tom there is some additional set up you will need to do:

- Add the bin directory to your path.
- Set the TOM_HOME environment variable to the directory where you installed Tom.
- JDK 1.4.2 or newer is needed. Optionally, set the JAVA_HOME environment variable. This should be set to the directory where your JDK is installed.

Windows Assuming Tom is installed in c:\jtom\. The following sets up the environment:

```
set TOM_HOME=c:\jtom
set PATH=%PATH%;%TOM_HOME%\bin
```

Unix Assuming Tom is installed in /usr/local/jtom. The following sets up the environment:

```
(bash)
  export TOM_HOME=/usr/local/jtom
  export PATH=${PATH}:${TOM_HOME}/bin

(csh)
  setenv TOM_HOME /usr/local/jtom
  set path=( $path ${TOM_HOME}/bin )
```

We also suggest you to define a TOM_LIB variable and update your CLASSPATH accordingly:

```
(bash)
  TOM_LIB=‘echo ${TOM_HOME}/lib/*.jar | tr ’ ’ ’:’‘
  export TOM_HOME=/usr/local/jtom
  export CLASSPATH=${TOM_LIB}:${CLASSPATH}

(csh)
  set TOM_LIB=‘echo ${TOM_HOME}/lib/*.jar | tr ’ ’ ’:’‘
  setenv CLASSPATH ${TOM_LIB}:${CLASSPATH}
```

Note: we recommend to add current directory . in your CLASSPATH.

9.4 Eclipse plugin

The Tom eclipse plugin is available directly from the Eclipse platform via the update link (tom.loria.fr/eclipse-update).

From the Help menu, choose Help Content and look at the Tom Section. You will find there everything to use Tom in a very simple way.

9.5 Getting some examples

Many examples have been written to test or illustrate some particular behaviors. This collection of examples is available at the tom.loria.fr/download.php page.

The Tom and Gom compilers are written in Tom itself. The compiler is certainly the most complex program written in Tom. Most of the constructs provided by Tom are used and tested in the project itself.

Chapter 10

Using Tom

10.1 The Tom compiler

The Tom compiler is a non-intrusive pattern matching compiler. It takes a Tom program, combination of a host programming language (Java, C or Caml) extended by Tom constructs. Each Tom program can define its own data structures and manipulate them in order to write pattern-matching based algorithms.

10.2 Development process

As mentioned earlier, Tom generates host language code. The command tom invokes the compiler on a Tom source file and generates a new source file (with .java, .tom.c, or .tom.ml extension, depending on the compilation command-line). The generated code has to be compiled/interpreted by a tool which can handle the corresponding host language: javac for Java, cc for C and ocamlc for Caml.

A typical development cycle is the following:

- edit a Tom program (by convention, a Tom source code has the .t extension)
- run the Tom compiler on this file
- compile the generated file (`javac file.java`, or `cc file.tom.c`,...)
- execute the program (`java file`, or `a.out`,...)

As cc and javac, tom can compile several input files. By default, the generated files are saved in the same directory as the corresponding input files. When compiling a Java file (i.e. a Tom file where Java is the host language), Tom is smart enough to parse the package definition and generate the pure-Java file in the same package architecture. Similarly to `javac`, `tom` supports a `-d` option which allows the user to indicate where the files have to be generated. To be compatible with `cc` and classical `Makefile`, Tom also supports a `-o` option which can be used to specify the name of the generated file.

10.3 Command line tool

tom [options] filename[.t]

The command takes only one file argument, the name of the input file to be compiled. By convention the input file name is expected to be of the form *filename.t*, whatever the used host language is.

Options:

`-X <file>`	Define an alternate XML configuration file Allow to define the plugins instantiated by the Tom platform
`--cCode | -c`	Generate C code (default is Java) The output file has the extension .tom.c
`--camlCode`	Generate Caml code (default is Java)
`--compile`	Activate the compiler (by default)
`--destdir | -d`	Specify where to place generated files By default, Tom generates files close to the input file. However (like `javac`), this option allows to specify a directory where generated files have to be put. When compiling a Java file, Tom is smart enough to parse the package definition and generate the pure-Java file in the same package architecture.
`--eclipse`	Activate Eclipse mode
`--expand`	Activate the expander (by default)
`--help | -h`	Show the help Give a brief description of each available options
`--import <path> | -I`	Path to included files Even if inclusion can be performed using relative path, this option specifies list of path where Tom look for when an inclusion has to be done by %include co
`--intermediate | -i`	Generate intermediate files The compiler manipulates Abstract Syntax Trees. This option dumps the AST after each phase (parsing, checking, expansion, compilation) This option is useful to analyze the compilation process
`--jCode | -j`	Generate Java code (by default)
`--lazyType | -l`	Use universal type This option makes Tom using a less restrictive backend. In Java, the universal sort Object is used more often. This reduces static typing but allows to manipulate inherited data-structures
`--noDeclaration | -D`	Do not generate code for declarations Avoid multiple declaration of symbols when structures are inherited or when multiple inclusion of common data structures are performed
`--noOutput`	Do not generate code No output file is generated. It allows to see warnings and errors without generating the result
`--noSyntaxCheck`	Do not perform syntax checking
`--noTypeCheck`	Do not perform type checking
`--optimize | -O`	Optimize generated code Add an optimization phase to the compilation process. This removes unused variables and performs some inlining.
`--optimize2 | -O2`	Further optimize generated code (does not imply `-O`)
`--output | -o`	Set output file name By default, Tom infers the name of the generated file by replacing the initial file extension by .java, .tom.c or .tom.ml, depending on the chosen target language. This option allows to explicitly give the name of the generated file.
`--parse`	Activate the parser (by default)
`--pretty | -p`	Generate readable code with indentation By default, the generated code is synchronized with the source code. This simplifies error reporting but makes the code more difficult to read. This option beautifies the generated code.
`--protected`	Generate protected functions In Java, this option forces the generation of protected functions, instead of private for symbol manipulation functions
`--static`	Generate static functions In Java, this option forces the generation of static functions for symbol manipulation functions
`--verbose | -v`	Set verbose mode on Give duration information about each compilation passes
`--version | -V`	Print the version of Tom
`--wall`	Print all warnings Useful in debugging phase

10.4 Ant task

Tom provides an ant task for running the Tom compiler within the apache ant build system.

The Tom ant task is very close in use to the `javac` ant task. Since this task is not part of the official ant tasks, you have first to declare this task in your buildfile, in order to use it.

This is done by the following code:

```
<taskdef name="tom" classname="tom.engine.tools.ant.TomTask">
    <classpath refid="tom.classpath"/>
</taskdef>
```

where `tom.classpath` is a path reference containing all the jar's in Tom's distribution.

This task is used to produce Java code from Tom programs. A typical use of the tom ant task is :

```
<tom config="${stable.configfile}"
         srcdir="${src.src}"
         destdir="${src.gen}"
         options="-I ${src.mapping}">
    <include name="**/*.t"/>
</tom>
```

Here, we want to compile all Tom source files in `{src.src}`, having the generated code in `{src.gen}`, and we configure the compiler to use the `{stable.configfile}` config file, and pass the options we want, just as we do for Tom in command line.

Another use of this task is :

```
<tom config="${stable.configfile}"
         srcdir="${src.src}"
         outputfile="${src.gen}/jtom/parser/TomParser.jj"
         options="-I ${src.mapping} -I ${src.gen}/jtom/adt">
    <include name="**/TomParser.t"/>
</tom>
```

Here we compile only one Tom file, and specify directly the output file name we want to use.

The main usecases for the tom ant task can be found in Tom's own buildfile.

The Tom ant task takes the following arguments:

Attribute	**Description**	**Required**
srcdir	Location of the Tom files.	Yes, unless nested `<src>` elements are present
destdir	Location of the java files.	No
outputfile	Destination file to compile the source. This require to have only one source file.	No
config	Location of the Tom configuration file.	Yes
logfile	Which log file to use.	No
optimize	Indicates whether source should be compiled with optimization; defaults to `off`	No
stamp	Indicates whether the generated code should use stamps or not, to simulate concrete ADT. Defaults to `false`	No
failonerror	Indicates whether the build will continue even if there are compilation errors; defaults to `true`	No
options	Custom options to pass to the Tom compiler	No
verbose	Asks the compiler for verbose output; defaults to `no`	No
pretty	Asks the compiler to generate more human readable code; defaults to `no`	No
nowarn	Asks the compiler not to report all warnings; defaults to `no`	No
classpath	The classpath to use, given as a reference to a path defined elsewhere. This variable is **not** used currently.	No

Chapter 11

Using Gom

11.1 Command line tool

People interested in using Gom as a standalone tool can use the Gom command line interface.

gom [options] filename [... filename]

Options:

`-X <file>`	Defines an alternate XML configuration file
`--destdir <dir> | -d`	Specify where to place generated files
`--verbose | -v`	Display compilation information
`--debug | -vv`	Display debugging information
`--intermediate | -i`	Leave generated ADT file for all compilation phases
`--import <path> | -I`	Path for searching included modules
`--package <pack> | -p`	Specify a prefix package name (optional)
`--wall | -W`	Print warnings
`--version | -V`	Print version
`--help | -h`	Show this help

11.2 Ant task

Gom can also be used within ant, and has its own ant tasks.

To use the Gom ant task, you have to first declare it the same way as the Tom ant task.

```
<taskdef name="gom"
         classname="tom.gom.tools.ant.GomTask"
         classpathref="tom.classpath"/>
```

Once the task is defined, you can use them to compile a Gom file.

The Gom task need the user to specify the configuration file to use with the `config` attribute. The default configuration file for Gom is included in the Tom distribution as `Gom.xml`.

```
<gom config="${stable.gomconfigfile}"
     package="tom.gom.adt"
     destdir="${src.gen}">
  <include name="tom/gom/adt/*.gom"/>
</gom>
```

The Gom task takes the following arguments:

Attribute	Description	Required
config	Location of the Gom configuration file.	Yes
options	Custom options to pass to the Gom compiler.	No
package	Package for the generated Java files.	No
destdir	Location of the generated files.	No

The set of `.gom` files to compile is specified with a nested `include` element.

Chapter 12

Configuring your environment

12.1 Editor

12.1.1 Emacs

12.1.2 Vim

12.2 Shell

12.2.1 Zsh